UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4

Institut für Softwaretechnik

Java-Faktenextraktor für Gupro

Studienarbeit

vorgelegt von:

Arne Baldauf
abaldauf@uni-koblenz.de

Nicolas Vika
ultbreit@uni-koblenz.de

betreut von:

Prof. Dr. Jürgen Ebert, Institut für Softwaretechnik, Fachbereich 4
Dr. Volker Riediger, Institut für Softwaretechnik, Fachbereich 4

Koblenz, im Januar 2008

Erklärung

Ich versichere, dass ich die Kapitel 1, 6, 7, 8 sowie die Anhänge A, B, D der vorliegenden Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Ort, Datum

Unterschrift

Erklärung

Ich versichere, dass ich die Kapitel 2, 3, 4, 5, 9 sowie die Anhänge C, E, F der vorliegenden Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Ort, Datum

Unterschrift

Zusammenfassung

Gupro soll Entwicklern im Rahmen der Softwarewartung bei Verständnis und Analyse von Softwaresystemen behilflich sein. Gupro verwendet dazu TGraphen, welche eine erheblich bessere Analyse ermöglichen als die zugrundeliegenden Quelltexte. Diese TGraphen sind typisierte, attributierte, gerichtete und angeordnete Graphen. Darüberhinaus kann von der verwendeten Programmiersprache beliebig abstrahiert werden. Zur Erzeugung der TGraphen werden programmiersprachenspezifische Faktenextraktoren benötigt.

Im Rahmen der Studienarbeit von Arne Baldauf und Nicolas Vika wird unter der Leitung von Prof. Dr. Jürgen Ebert und Dr. Volker Riediger ein Faktenextraktor für Java-Quelltexte entwickelt. Der Javaextraktor generiert eine TGraph-Repräsentation der Quelltexte eines in Java implementierten Softwareprojekts. Der erzeugte TGraph ist dann durch die Werkzeuge des Gupro-Projekts weiter verarbeitbar.

Der Javaextraktor parst Javaquelltexte bis einschließlich Javaversion 6 und setzt diese in TGraphrepräsentationen um. Dazu wird im Rahmen der Studienarbeit ein Werkzeug gesucht, welches den Parsingvorgang übernimmt. Zusätzlich wird ein Javametamodell und entsprechendes Schema für den TGraph entwickelt. Metamodell und Schema geben Typisierung, Attributierung und Struktur des TGraphen vor.

Inhaltsverzeichnis

1 Einführung und Motivation

Software muss nach ihrer Auslieferung gewartet werden. Das heißt, sie muss korrigiert, modifiziert, weiterentwickelt oder perfektioniert werden. Dieser Prozess kann Jahre oder auch Jahrzehnte über den ursprünglichen Auslieferungstermin hinaus andauern.

Es existieren viele Altsysteme, welche heute noch im Produktiveinsatz sind und weiterhin gewartet werden müssen. Oft jedoch sind die ursprünglichen Entwickler nicht mehr für die Software zuständig. Hinzu kommt nicht selten ein unzureichender Dokumentationsgrad der Software. Diejenige Person, die nun in die Wartung des Systems involviert wird, befindet sich somit meist in einer schwierigen Ausgangssituation.

An dieser Stelle setzen die Gupro-Tools zur Unterstützung von Reengineering und Programmverstehen an. Sie sollen den Wartungsingenieur dabei unterstützen, den Quelltext des Softwaresystems zu erschließen.

Die Gupro-Tools arbeiten dabei auf Graphrepräsentationen des Quelltextes, welche dazu von programmiersprachenspezifischen Extraktoren aus dem Quelltext generiert werden. Bisher existieren nur Extraktoren für ältere Programmiersprachen.

Ziel der vorliegenden Studienarbeit war es, einen Extraktor für Java-Code bis zur aktuellen Version 5[1] zu realisieren. Dessen Entwicklung und Verwendung wird in diesem Dokument beschrieben, wobei der Extraktor im Folgenden auch als Javaextraktor oder Faktenextraktor bezeichnet wird.

Diese Studienarbeit richtet sich an Leser, die mindestens über den Wissensstand eines Informatikstudenten mit abgeschlossenem Vordiplom verfügen.

1.1 Erklärung der Begrifflichkeiten

In diesem Abschnitt werden Begriffe definiert und erläutert, die in dieser Arbeit häufiger Verwendung finden.

TGraph. Jeder Graph besteht aus Knoten und Kanten. Ein TGraph hat zusätzliche Eigenschaften, er ist

- typisiert: Knoten und Kanten haben einen bestimmten Typ.
- attributiert: Knoten und Kanten können beliebig viele Attribute besitzen (somit auch keine).

[1]Im Verlauf der Studienarbeit erschien Java 6, dieses besitzt im Vergleich zur Vorgängerversion keine neuen Sprachkonstrukte, so dass die gesamte Funktionalität des Javaextraktors auch diese Version abdeckt.

- gerichtet: Kanten besitzen eine Richtung.
- angeordnet: die Menge, der mit einem Knoten verbundenen Kanten besitzt eine Ordnung (1., 2., ..., n-te Kante).

Desweiteren gilt, dass eine Kante nur zwischen zwei Knoten aufgespannt sein darf. TGraphen können zyklisch sein. Eine formale Definition und weitergehende Beschreibung findet sich auf den Seiten 52-54 in [6].

GReQL. GReQL steht für Gupro Repository Query Language und ist eine Anfragesprache für TGraphen, die in Gupro Verwendung findet. Abgefragt werden können:

- Inhalte, z. B. Attributwerte von Kanten und Knoten
- Strukturen im TGraph
- Aggregierte Informationen, z. B. Anzahl von Knoten oder Kanten eines bestimmten Typs
- beliebige Kombinationen aus den drei zuvor genannten

Für die Formulierung einer Anfrage ist die Kenntnis des Schemas / Metamodells des angeftragten TGraphen notwendig. Details zu GReQL finden sich u. a. auf den Seiten 173ff in [6] sowie für die aktuelle Version GReQL 2 in [3] und [12].

JGraLab Das zuvor in C++ implementierte Graphenlabor wurde in Java als JGralab neuimplementiert und ist Teil des Gupro-Projekts. „Das Graphenlabor bietet eine Klassenbibliothek, um TGraphen als interne Datenstruktur effizient benutzen zu können. Diese API bietet Traversierungs- und Manipulationsmöglichkeiten für TGraphen und ihr Typsystem an“ (vgl.: Seite 5 in [11]). Weitere Details zu JGraLab finden sich ebenfalls in [11].

Sourcecodebrowser Der Sourcecodebrowser ist eine Komponente der QGupro Benutzeroberfläche. Er zeigt Quelltexte an und hebt Codeabschnitte hervor, die das Ergebnis einer GReQL-Anfrage sind. Für mehr Informationen zum Sourcecodebrowser siehe [2].

Token. Ein Token ist ein atomares Eingabeelement für einen Parser und wird von einem Lexer erzeugt. Es repräsentiert eine erkannte Zeichenfolge in einem Zeichenstrom (bspw. Quelltext). Üblicherweise wird ein Token in einer Lexergrammatik definiert.

Lexer. Ein Lexer ist ein Programm, welches aus einem Zeichenstrom eine Folge von Tokens erzeugt. Der Lexer ist in der Lage bestimmte Zeichen oder Zeichenfolgen zu erkennen. Diese werden als Lexeme bezeichnet. Aus jedem erkannten Lexem erzeugt der Lexer ein Token.

Ein Lexer kann durch eine formale Grammatik beschrieben werden. Aus dieser kann automatisiert ein Lexer erzeugt werden.

Parser. Ein Parser ist ein Programm zur Syntaxanalyse. Der Parser erkennt syntaktische Einheiten in einem Tokenstrom. In der Regel liefert ein Lexer die Tokens. Meist bildet der Parser die Syntax der Eingabedaten in einem abstrakten Syntaxbaum ab.

Auch ein Parser kann durch eine formale Grammatik beschrieben werden. Aus dieser kann wiederum automatisiert ein Parser erzeugt werden.

LL(*)-Parser. Bezeichnung für einen Parsertyp. LL(*)-Parser versuchen die Syntax mit einem Top-down-Ansatz zu erkennen. Der Parser kann zur Entscheidungsfindung beliebig viele Tokens weit vorausschauen, dieser Wert wird als Lookahead bezeichnet. Eine formale Definition und Beschreibung von LL-Parsern findet sich in [1] auf den Seiten 174ff und 190-191.

Treewalker. Ein Treewalker ist ein Programm, welches einen Baum traversiert. Im Rahmen der Studienarbeit werden aus Javaquelltexten abstrakte Syntaxbäume von einem Parser erzeugt und von einem Treewalker traversiert.

Auch ein Treewalker kann durch eine Grammatik beschrieben werden. Aus dieser kann wiederum automatisiert ein Treewalker erzeugt werden.

Abstrakter Syntaxbaum (AST). Ein AST ist ein Baum, der durch seine Struktur die Syntax eines Textes abbildet. Im Rahmen der Studienarbeit sind dies Javaquelltexte.

Abstrakter Syntaxgraph (ASG). Im Gegensatz zu einem abstrakten Syntaxbaum darf ein ASG zyklisch sein und identische Elemente durch einen einzelnen Knoten oder Blatt repräsentieren.

Quelltextreferenz. Quelltextreferenzen sind Typspezifikationen, Methoden- oder Konstruktoraufrufe, sowie Zugriffe auf Felder, Variablen, Parameter und Enum-Konstanten. Dies sind allesamt Elemente im Quelltext, welche sich auf ein an anderer Stelle deklariertes Element beziehen. Z. B. ist ein Variablenzugriff durch den Variablenamen im Quelltext eine Quelltextreferenz. Der Name referenziert die Variable. Grundlage der Variable ist jedoch ihre konkrete Deklaration an einer anderen Stelle im Quelltext. Es besteht somit eine semantische Beziehung zwischen Benutzung und Deklaration des selben Elements.

Nicht gemeint ist die in objektorientierten Programmiersprachen übliche Objektreferenz, die zum Beispiel beim Übergeben von Argumenten verwendet wird.

2 Anforderungsliste Javaextraktor

Zu Beginn der Studienarbeit wurden die Anforderungen an den Javaextraktor aufgestellt. Zum Teil stammen diese unmittelbar aus der Aufgabenstellung, teilweise ergeben sie sich indirekt aus dem Kontext der Verwendung des Javaextraktors und manche sind eine Festlegung einer allgemein bekannten, geeigneten Vorgehensweise.

Die Anforderungen an den Javaextraktor wurden mit Prioritäten versehen. Dabei entsprechen die verwendeten Farben den Prioritäten für die Umsetzung der jeweiligen Anforderung.

- MUSS - Die Anforderung muss auf jeden Fall umgesetzt werden.
- SOLLTE - Die Anforderung sollte umgesetzt werden, wenn es möglich ist.
- OPTIONAL - Die Anforderung ist optional.

Ferner ist nach jeder Anforderung eine kurzer Vermerk über den Status ihrer Umsetzung aufgeführt.

2.1 Funktionale Anforderungen

Die funktionalen Anforderungen legen fest, was der Javaextraktor leisten soll.

1. MUSS - Der Faktenextraktor soll Java-Quelltexte, nach Möglichkeit bis einschließlich Version 5, zu einem TGraphen umsetzen können.
   Erfüllt.
2. OPTIONAL - Die Schemata sollen, neben einer feinkörnigen Graphenrepräsentation des Codes, auch höhere Abstraktionsniveaus abdecken.
   Partiell erfüllt - nur das feingranulare Schema wurde umgesetzt.
3. SOLLTE - Der Faktenextraktor soll als eigenständiges Programm lauffähig sein, nach Möglichkeit als Commandline-Utility.
   Erfüllt.
4. OPTIONAL - Für die Arbeit mit TGraphen sollen einige sinnvolle Beispielanfragen in GReQL formuliert werden, welche auf die objektorientierte Ausrichtung von Java abzielen.
   Nicht erfüllt.
5. MUSS - Die Positionen der identifizierten Elemente im Quelltext müssen Gupro-konform im TGraph mitgespeichert werden, zwecks Nutzung durch den graphenbasierten Sourcecodebrowser.
   Erfüllt.

6. SOLLTE - Der Faktenextraktor soll möglichst fehlertolerant sein. D.h. syntaktisch fehlerhafter Java-Code sollte (durch Ignorieren oder Auslassen) nicht zum Fehlschlagen des kompletten Parsingvorgangs führen.
   Nicht erfüllt. Die verwendete Javagrammatik ließ dies nicht zu.

7. MUSS - Der Faktenextraktor muss Quelltextreferenzen auflösen können.
   Erfüllt.

8. OPTIONAL - Der Faktenextraktor soll nur Code feingranular im TGraph repräsentieren, welcher als Quellcode vorliegt; von referenziertem Code (welcher nur in .class- oder in .jar-Dateien ohne Quelltext vorliegt) müssen mittels Reflexion lediglich die Signaturen extrahiert werden.
   Erfüllt.

2.2 Nichtfunktionale Anforderungen

Die nichtfunktionalen Anforderungen legen die weiteren Eigenschaften des Javaextraktors fest.

9. MUSS - Der Faktenextraktor soll effizient arbeiten. Konkret bedeutet dies, dass der Extraktor eine - in Abhängigkeit von der Menge des zu analysierenden Quelltextes - vertretbare Laufzeit aufweisen soll.
   Erfüllt.

10. OPTIONAL - Der Faktenextraktor soll in Java realisiert werden.
    Erfüllt.

11. MUSS - Der Programmierstil aus der Vorlesung „Programmierung“ soll angewendet werden.
    Erfüllt.

12. OPTIONAL - Der Quelltext soll mit Unit-Testing überprüft werden.
    Nicht erfüllt.

13. MUSS - Der Quelltext soll Javadoc-kompatibel kommentiert werden.
    Erfüllt.

14. MUSS - Die Architektur soll in UML visualisiert werden.
    Erfüllt.

Ferner sollte nach bestehenden Tools recherchiert werden, welche für den Javaextraktor verwendet werden können. Folgende Anforderungen ergaben sich dadurch:

15. SOLLTE - Bestehende Werkzeuge sollten den eigenen Entwicklungsaufwand soweit wie möglich reduzieren.
    Erfüllt. Es war keine Entwicklung eines eigenen Lexers und Parsers nötig, da diese automatisch aus einer bereits existierrenden Grammatik erzeugt werden konnten.

16. SOLLTE - Bestehende Werkzeuge sollten sich möglichst noch in der aktiven Weiterentwicklung befinden, so dass evtl. bestehende Fehler korrigiert werden und eine Weiterentwicklung (etwa für neue Java-Versionen) absehbar ist.
    Erfüllt. Bei ANTLR ist dies der Fall.

17. MUSS - Bestehende Werkzeuge sollen hinsichtlich ihrer Tauglichkeit für dieses Projekt bewertet werden.
    Erfüllt. Bewertung abgeschlossen.

18. MUSS - Recherchierte Werkzeuge sollten, selbst im Falle der Unbrauchbarkeit für dieses Projekt, kurz beschrieben und abgehandelt werden.
    Erfüllt. Abhandlungen erstellt.[2]

[2] Für eine detaillierte Ausführung zur Onlinerecherche siehe Anhang C

3 Onlinerecherche nach einem Javaparser

Zur Extraktion von Fakten aus Javaquelltexten müssen sie geparst werden. Parsingtechniken werden schon lange im Compilerbau verwendet und stehen in einer Vielzahl von bereits existierenden Werkzeugen zur Verfügung. Deshalb wurde zunächst eine umfangreiche Onlinerecherche vorgenommen. Ziel war es ein bereits existierendes Werkzeug zu finden, welches bei der Erfüllung der Anforderungen aus dem letzten Kapitel von Nutzen sei und auf dessen Basis der Javaextraktor aufgebaut werden konnte.

Dieses Kapitel beschreibt zunächst Kriterien, anhand derer die Funde bewertet wurden und schließt mit einer Aufstellung der Ergebnisse ab. Eine detaillierte Ausführung der Onlinerecherche ist in Anhang C zu finden .

3.1 Bewertungskriterien für die gefundenen Werkzeuge

Jedes gefundene Werkzeug wurde auf die folgenden Eigenschaften hin untersucht:

Parser in Java. Zur Einbindung in den in Java zu entwickelnden Javaextraktor wäre ein javabasierter Parser ideal. Ferner wäre der Javaextraktor dadurch plattformunabhängig.

Ist ein Werkzeug nicht in Java geschrieben, führt dies jedoch nicht automatisch zum Ausschluss, da keine Festlegung auf eine einzige Programmiersprache besteht.

Grammatik für Java bis Version 5. Damit der Javaextraktor auch aktuelle Softwareprojekte verarbeiten kann, sollte das Parsen von Quelltexten bis Javasprachversion einschließlich Version 5 möglich sein. Neben der Recherche in der Projektdokumentation wurde dies auch mittels Parsingtests mit Quelltexten, die spezifische Sprachkonstrukte aus Java 5 enthielten, überprüft.

Besteht ein Werkzeug diesen Test nicht, so führte dies noch nicht zum Ausschluss, da dieser Mangel eventuell behoben werden kann.

Verfügbarkeit der Quelltexte. Gerade für die Verwendung im Javaextraktor ist eine Verfügbarkeit der Quelltexte zwingend erforderlich. Auch die Lizenzierung von Quelltexten gegen vertretbare Kosten ist möglich.

Ist dies nicht der Fall, führt dies zum Ausschluss des Werkzeuges.

Dokumentationsgrad. Umfasst die Dokumentation zumindest ein Benutzerhandbuch, welches die Verwendung und ggf. Erzeugung des Werkzeuges erklärt, wird der Dokumentationsgrad als *niedrig* bewertet. Gibt es zusätzlich eine API-Spezifikation, so ist der Dokumentationsgrad *mittel*. Existieren darüberhinaus FAQ(s), Tutorial(s), Wiki(s) oder vergleichbares, gilt der Dokumentationsgrad als *hoch*. Ist desweiteren auch noch Support durch Mailinglisten oder Foren gegeben, dann ist der Dokumentationsgrad als *sehr hoch* bewertet.

Existiert *keine* brauchbare Dokumentation, führt dies zum Ausschluss des Werkzeuges.

Aktivität. Befand sich das Werkzeug in einer ständigen Weiterentwicklung und Verbesserung und existierte eine lebendige Community um dieses Projekt, war dies als positiv zu bewerten - gerade im Hinblick auf eine durchaus zu erwartende spätere Weiterentwicklung des Javaextraktors.

War dies nicht der Fall, führte dies nicht unmittelbar zum Ausschluss des Werkzeuges.

Position und Länge eines Elements im Quelltext abrufbar. Zur weiteren Verwendung der erzeugten Graphen in Gupro müssen Position und Länge von Elementen im Quelltext im Graphen gespeichert werden. Dazu sollte das Werkzeug die Abfrage dieser Werte ermöglichen.

Ist dies nicht der Fall, führt dies nicht unmittelbar zum Ausschluss des Werkzeuges, da diese Funktionalität eventuell nachgerüstet werden kann.

3.2 Ergebnis der Onlinerecherche

Untersucht wurden Parsergeneratoren, Werkzeuge, die Parsergeneratoren benutzen, quelloffene Java-Compiler-Implementationen[3] und die Java Development Tools aus Eclipse.

Tabelle 1 listet alle untersuchten Werkzeuge und die ermittelten Ergebnisse auf. Bei Werkzeugen, die nicht zum Laufen gebracht werden konnten oder die laut Dokumentation keine Unterstützung von Java 5 bieten, wurden keine Parsingtests durchgeführt. Dies ist entsprechend in der Tabelle vermerkt (-). Erfüllte ein Werkzeug mehr als ein Kriterium nicht, so wurde es ausgeschlossen. Kriterien, die zum Auschluss führten, sind in der Tabelle fett geschrieben. Werkzeuge, die in die nähere Auswahl kamen, stehen in einer grau hinterlegten Zeile.

Keines der betrachteten Werkzeuge kann alle geforderten Kriterien vollständig erfüllen. In die nähere Auswahl kamen sechs Werkzeuge. GCJ wurde ausgeschlossen, da genügend javabasierte Lösungen zur Auswahl standen. Die Grammatik von JavaCC hätte korrigiert und jene

[3]Zum Zeitpunkt der Onlinerecherche war Suns Javacompiler `javac` noch nicht als Open Source freigegeben. Daher wurde dieser nicht untersucht.

Werkzeug	**Parser in Java**	**Parsing von Java bis Version**	**Parsingtests bestanden**	**Quelltexte frei verfügbar**	**Dokumentation**	**aktives Projekt**	**Position und Länge eines Token abrufbar**
ANTLR	ja	5	ja	ja	sehr hoch	ja	ja
JReFactory	ja	5	-	ja	**keine**	**nein**	ja
JavaCC	ja	5	nein	ja	hoch	ja	ja
CUP	ja	5	**nein**	ja	niedrig	ja	**nein**
Coco/R	ja	1.4	-	ja	mittel	ja	ja
Cocktail	ja	5	**nein**	**nein**	niedrig	ja	ja
Jabstract	ja	**1.1**	-	ja	niedrig	**nein**	ja
FUJABA	ja	1.4	-	ja	sehr hoch	ja	ja
GCJ	nein	5	ja	ja	hoch	ja	?
Java Espresso	ja	**1.0**	-	ja	mittel	**nein**	?
JDT	ja	5	ja	ja	hoch	ja	?

Tabelle 1: Übersicht der untersuchten Werkzeuge

von FUJABA und Coco/R für Java 5 aktualisiert werden müssen. Bei den Java Development Tools schien eine Trennung von Eclipse nur sehr schwer möglich - der Javaextraktor hätte als Eclipse-Plugin realisiert werden müssen.

Aufgrund des geschätzten Anpassungsaufwands im Vergleich zu anderen Werkzeugen fiel die Wahl schließlich auf ANTLR. Bei ANTLR war es lediglich notwendig, die Funktion zur Positionsabfrage zu modifizieren. Hinzu kam, dass von allen untersuchten Werkzeugen das ANTLR-Projekt die mit Abstand aktivste Community vorweisen konnte. Das nächste Kapitel widmet sich der näheren Beschreibung von ANTLR.

4 ANTLR

Dieses Kapitel behandelt das Werkzeug „ANother Tool for Language Recognition“, kurz ANTLR. Zunächst erfolgt ein Überblick über das Projekt, gefolgt von einem Abschnitt über die Funktionsweise. Zuletzt werden die Anpassungen, die zur Erfüllung aller gestellten Anforderungen und Integration in den Javaextraktor nötig waren, beschrieben.

4.1 Überblick

Historie. Die Entwicklung von ANTLR begann 1988, ursprünglich als Projekt an der Purdue Universität (West Lafayette, Indiana, USA), damals noch unter den Namen „YUCC“ bzw. „PCCTS“. Das Projekt wurde 1989 Inhalt der Masterarbeit von Terence Parr, welcher seitdem die Schlüsselfigur hinter ANTLR ist.

Die erste öffentlich verfügbare Version erschien 1990 und ist von jeher quelloffen. Das Projekt wurde kontinuierlich weiterentwickelt und Terence Parr beschäftigt sich heute noch, als Professor an der Universität von San Francisco, mit ANTLR. Aktuell ist Version 3. Der Javaextraktor basiert jedoch auf Version 2.7.6, da Version 3 erst während der Implementierung des Javaextraktors fertig gestellt wurde und noch keine zu ANTLR 3 kompatible Grammatik für Java[4] existierte.

Charakterisierung. ANTLR ist ein Framework zur Erstellung von Compilern bzw. deren einzelner Bestandteile, mit besonderem Schwerpunkt auf der Unterstützung von abstrakten Syntaxbäumen (Erzeugung, Traversierung, Konvertierung). ANTLR liegt selbst in verschiedenen Sprachen vor und kann LL(*)-Parser in Java, C++ und C# erzeugen. Dabei unterstützt ANTLR die automatische Erzeugung von abstrakten Syntaxbäumen. Ein Treewalker zu deren Traversierung kann ebenfalls aus einer separaten (zur eigentlichen Grammatik kompatiblen) Baumgrammatik generiert werden.

Hinzu kommt eine graphische Benutzeroberfläche (ANTLRWorks) zum Entwickeln, Debuggen und Testen von Grammatiken. Zusätzlich existiert eine graphische Oberfläche für die Benutzung der Kommandozeilenwerkzeuge unter Eclipse, ANTLR Studio genannt. Allerdings stammt es von einem Dritten und ist kostenpflichtig.

Für Java 5 und ANTLR v2.7.6 sind drei verschiedene Grammatiken verfügbar. Zusätzlich existieren Grammatiken für eine Vielzahl weiterer Sprachen.

[4] Die Syntax der Grammatiken für ANTLR Version 2 und Version 3 unterscheidet sich.

Literatur und Support. Auf der ANTLR Homepage [13] steht eine umfangreiche Dokumentation, bestehend aus einer FAQ, vielen Tutorials, einem Benutzerhandbuch, einem Wiki, einer API-Beschreibung des Frameworks und zusätzlichen Veröffentlichungen von, teils wissenschaftlichen, Arbeiten zum Thema Compilerbau, zur Verfügung. Fragen, die darüber hinausgehen, können in Foren sowie in einer Mailingliste gestellt werden.

Rund um dieses Projekt existiert eine große und aktive Community. Dieser entstammen u. a. die Grammatiken für Java 5.

4.2 Funktionsweise von ANTLR

Der Parsergenerator von ANTLR erzeugt aus einer Grammatik einen Parser. Im Gegensatz zu den meisten anderen Generatoren kann ANTLR jedoch ohne Umwege auch (zum Parser passende) Lexer und Treeparser erzeugen. Es ist dazu nicht nötig, weitere Zusatzprogramme zu Hilfe zu nehmen, wie es beispielsweise bei CUP [25] der Fall ist[5].

Grundsätzlich gilt, dass für Lexer, Parser und Treeparser jeweils eine Grammatik geschrieben werden muss. Dabei kann der Benutzer frei wählen, welche er realisieren will. Braucht er beispielweise nur einen Lexer, so muss er nur die Lexergrammatik schreiben. Da ANTLR dem Paradigma der Objektorientierung folgt, ist es auch möglich, die Grammatiken weiter zu vererben.

Bei allen im Framework enthaltenen Werkzeugen handelt es sich um Kommandozeilenprogramme. Zur Erzeugung der gewünschten Programme müssen die Grammatiken per Kommandozeilenparameter an ANTLR übergeben werden. Die so erzeugten Lexer und Parser sind im Quelltext vorliegende Komponenten und können zur Verwendung in separaten Programmen eingebunden werden. Dabei werden zur Laufzeit weiterhin die Bibliotheken von ANTLR benötigt.

Umfangreicher und detaillierter geht Anhang E auf die Funktionsweise von ANTLR ein.

4.2.1 Javagrammatik für ANTLR

Für den Javaextraktor wurde die Javalösung von Michael Studman [18] ausgewählt, da sie Grammatiken für den Lexer, Parser und Treewalker umfasst und die daraus erzeugten Programme in den Parsingtests am besten abschnitten. Zusätzlich kann der erzeugte Parser die Quelltexte des Java Developer Kit Version 1.5_06, Eclipse und ANTLR Version 2.7.6 fehlerfrei parsen. Mit hoher Sicherheit kann behauptet werden, dass gültiger Javacode[6] zuverlässig erkannt wird.

[5] CUP kann keine „eigenen" Lexer erzeugen. Es muss zunächst ein passender Lexer geschrieben oder mit einem Scannergenerator wie JLex [21] erzeugt werden.

[6] Kompilierbar durch `javac`.

Grundsätzlich können alle Quelltexte geparst werden, die in ASCII, ANSI (mit verschiedenen Codepages) und Unicode (UTF-7, UTF-8, Unicode, Big Endian) codierten Dateien vorliegen. Nur Quelltexte, die Bezeichner mit Sonderzeichen oder Unicode-Escapesequenzen[7] enthalten, können nicht erfolgreich geparst werden[8].

Zum Parsen der Javaquelltexte sollte der erzeugte Parser verwendet werden. Die Erzeugung der TGraphrepräsentation sollte dann durch den Treewalker während der Traversierung der abstrakten Syntaxbäume geschehen.

4.3 Nötige Anpassungen

Vor der Implementierung des eigentlichen Javaextraktors waren noch Anpassungen notwendig, um die erwünschten Funktionen für Lexer und Parser vollständig zu realisieren. Die folgenden Abschnitte beschreiben diese Anpassungen.

4.3.1 Anpassung Position des Tokens als Offset

Die Position eines Tokens im Quelltext lässt sich mit der von ANTLR bereitgestellten Funktionalität nicht als Offset abrufen. Lediglich Zeile, Spalte und Länge (über die Länge des Token-Strings) des Tokens sind einsehbar. Um diese Funktionalität nachzurüsten, muss erst die interne Funktionsweise von ANTLR verstanden werden.

Tokens werden in ANTLR durch die Klasse `ANTLR.CommonToken` implementiert. Diese werden von einem Lexer erzeugt, der wiederum aus einer Grammatik erzeugt wird. In der Regel erben die Lexer von der Klasse `ANTLR.CharScanner`. Diese steuert die Klasse `ANTLR.LexerSharedInputState`, welche die aktuelle Position im Eingabestrom kennt und diese in die Tokens schreibt.

Um den Offset nachzurüsten, wurden Adapterklassen implementiert, die von den o. a. Klassen aus ANTLR erben. Die neue Tokenklasse `CommonTokenAdapter` hat nun zusätzliche Methoden zum Speichern und Abrufen der Position als Offset. Der neue `LexerSharedInputStateAdapter` kann mit den neuen Tokens umgehen. Der neue Lexer `CharScannerAdapter` kann den neuen `LexerSharedInputStateAdapter` nutzen und auch mit den neuen Tokens umgehen. Da dieser aus der Grammatik in `java.g`[9] erzeugt wird, musste diese angepasst werden. Durch Angabe von

```
1 class JavaLexer extends Lexer( "javaextractor.adapters.CharScannerAdapter" );
```

[7]Unicode-Escapesequenzen in Java besitzen das Format `\uxxxx`, wobei `xxxx` der Hexadezimalcode des Zeichens ist, z.B. \u00e4 für „ä“.
[8]Für detaillierte Aufstellung der Ergebnisse der Parsingtests siehe Anhang C und D.
[9]im Package `javaextractor`

antlr::CharScanner

antlr::LexerSharedInputState

CharScannerAdapter

+ CharScannerAdapter(LexerSharedInputState)
+ CharScannerAdapter(InputBuffer)
+ getColumn() : int
+ getLine() : int
+ getOffset() : int

LexerSharedInputStateAdapter

- file: String
+ offset: int = 0

+ annotate(CommonTokenAdapter) : void
+ getFileName() : String
+ getOffset() : int
+ LexerSharedInputStateAdapter(InputBuffer)
+ LexerSharedInputStateAdapter(String, InputStream)
+ LexerSharedInputStateAdapter(String)
+ reset() : void
+ setColumn(int) : void

Wird durch ANTLR aus java.g erzeugt

javaextractor::JavaLexer

JavaLexerAdapter

+ JavaLexerAdapter(LexerSharedInputState)
makeToken(int) : Token
+ newline() : void
+ tab() : void

«controls»

1

1

Abbildung 1: Klassendiagramm, Auszug aus Package `javaextractor.adapters`

zu Beginn der Lexergrammatik dient `CharScannerAdapter` nun als Basistyp für den erzeugten Lexer. Damit der Offset richtig berechnet wird, erbt der `JavaLexerAdapter` vom erzeugten Lexer und überschreibt die Methoden `newline()` und `tab()` [10].

Um die neue Funktionalität zu nutzen, muss ein `LexerSharedInputStateAdapter` instanziiert werden und bei der Instanziierung des `JavaLexerAdapter` im Konstruktor übergeben werden. Der Lexer muss wiederum bei der Instanziierung des Parsers im Konstruktor übergeben und angewiesen werden, die neuen Tokens zu erzeugen. Dies kann auf folgende Weise geschehen:

```
1 LexerSharedInputStateAdapter inputState = new LexerSharedInputStateAdapter(
      fileName );
2 JavaLexerAdapter javaLexer = new JavaLexerAdapter( inputState );
3 javaLexer.setTokenObjectClass( "javaextractor.adapters.CommonTokenAdapter" );
4 JavaRecognizer javaParser = new JavaRecognizer( javaLexer );
```

Abbildung 1 veranschaulicht die Anpassung in einem Klassendiagramm.

4.3.2 Anpassung Position des Tokens im AST verfügbar

Hinzu kommt, dass diese Informationen nicht mehr im erzeugten AST abrufbar sind. Um die Position eines Tokens selbst im AST verfügbar zu machen, waren zusätzlich Maßnahmen nötig. Abstrakte Syntaxbäume werden in ANTLR durch die Klasse `ANTLR.CommonAST` implementiert. Diese werden durch einen Parser erzeugt. Die Attribute des AST werden durch `initialize`-Methoden gefüllt. Wenn der Parser ein AST-Element erzeugt, ruft er zusätzlich eine dieser Methoden auf.

Um die Position eines Tokens selbst im AST verfügbar zu machen, wurde wieder eine Adapterklasse implementiert. Die neue Klasse `CommonASTAdapter` kennt zusätzlich „ihr“ Token und überschreibt alle `initialize`-Methoden, damit sie die Referenz auf das Token setzen kann.

Um die neue Funktionalität zu nutzen, muss der Parser angewiesen werden, die neuen AST-Elemente zu erzeugen. Zusätzlich zu o. a. Anweisungen aus Listing 4.3.1 reicht dazu die Angabe von:

```
1 javaParser.setASTNodeClass( "javaextractor.adapters.CommonASTAdapter" );
```

Abbildung 2 veranschaulicht die Anpassung in einem Klassendiagramm.

[10] Erhöht den Offset nur um 1, wenn ein Tabulator im Eigabestrom erkannt wird.

antlr::CommonAST

antlr::CommonToken

CommonASTAdapter

+ CommonASTAdapter()
+ CommonASTAdapter(Token)
+ getColumn() : int
+ getLine() : int
+ getOffset() : int
+ getText() : String
+ getToken() : Token
+ getType() : int
+ initialize(Token) : void
+ initialize(String, int) : void
+ initialize(AST) : void
+ setText(String) : void
+ setToken(Token) : void
+ setType(int) : void

knows

1

1

CommonTokenAdapter

endCol: int
endLine: int
offset: int

+ CommonTokenAdapter()
+ CommonTokenAdapter(String)
+ CommonTokenAdapter(int, String)
+ getEndColumn() : int
+ getEndLine() : int
+ getFileName() : String
+ getOffset() : int
+ setEndColumn(int) : void
+ setEndLine(int) : void
+ setFileName(String) : void
+ setFromCommonASTAdapter(CommonASTAdapter) : void
+ setOffset(int) : void

Abbildung 2: Klassendiagramm, Auszug aus Package `javaextractor.adapters`

4.3.3 Anpassungen zum Aufnehmen der Kommentare in den TGraph

Mit der ursprünglichen Grammatik wurden während des Parsens alle im Quelltext enthaltenen Kommentare korrekt erkannt, jedoch direkt verworfen[11]; d. h. Kommentare werden somit nicht im abstrakten Syntaxbaum abgebildet. Da aber auch Kommentare mit in den Graph aufgenommen werden sollten, mussten entsprechende Anpassungen vorgenommen werden. Außerdem sollte zwischen einzeiligen, mehrzeiligen und Javadoc-Kommentaren unterschieden und eine entsprechende Repräsentation im Graphen erzeugt werden. Da nicht immer festgestellt werden kann, welchem Element ein Kommentar im Quelltext zugeordnet ist, sollten alle Kommentare mit dem obersten Repräsentanten einer Quelltextdatei im TGraphen (`TranslationUnit`) verbunden werden.

Da der Graph beim Durchlaufen des abstrakten Syntaxbaums erzeugt werden soll, wäre eine Abbildung der Kommentare in diesen nötig gewesen. Dazu hätten alle drei Grammatiken in größerem Maße angepasst werden müssen, da Kommentare fast an jeder beliebigen Stelle[12] in Javaquelltexten stehen dürfen. Deshalb wurde nur die Lexergrammatik modifiziert. Der Lexer erzeugt keine Tokens, die Kommentare repräsentieren. Jedoch werden nun alle Kommentare einer Datei im Lexer gesammelt und stehen nach dem Parsen der Datei als Liste zur Verfügung.

[11] Da sie für die Funktion des Programms unerheblich sind.
[12] Kommentare sind in Java überall dort möglich, wo ein Leerzeichen stehen darf.

Die Kommentare können dann per `getComments()` abgefragt und in den Graph eingebaut[13] werden. Die Grammatiken von Parser und Treewalker blieben unverändert, da diese von dem so geschaffenen Mechanismus nicht tangiert werden.

4.3.4 Fehlende Quelltextelemente

Neben Kommentaren wurden in der ursprünglichen Grammatik auch weitere Elemente aus dem Quelltext nicht in den AST aufgenommen. Diese Elemente werden im Extraktor benötigt, um die Länge von allen Javasprachkonstrukten berechnen zu können[14] (z. B. für den Rumpf einer Klasse). Dabei handelte es sich um Semikola, Klammern und bislang fehlende Schlüsselworte.

Entsprechend wurden die Parser- und Lexergrammatiken erweitert, so dass diese Elemente nun in den AST aufgenommen und vom Treewalker korrekt behandelt werden können.

[13]Siehe dazu Kapitel 6.3.
[14]Siehe dazu Kapitel 7.2.

5 Metamodell für Java 5

Der Javaextraktor erzeugt einen TGraphen. Dieser wird durch ein Metamodell beschrieben. Es wird deshalb eines für den Javaextraktor benötigt. Dieses Kapitel definiert zunächst die Begriffe Metamodell und Schema und beschreibt anschließend die Anforderungen an das spezifische Metamodell für Java 5.

5.1 Begriffsdefinitionen

Metamodell. Ein Modell beschreibt die Menge aller seiner möglichen Instanzen. Ein Metamodell beschreibt, wie ein Modell aussehen kann und somit die Menge aller seiner möglichen Instanzen (welche Modelle sind).

Im Sinne der Studienarbeit ist jeder Graph, der aus einem Quelltext extrahiert wird, ein Modell dieses Quelltextes. Ein Graph besteht aus Knoten und Kanten. Wie diese typisiert, attributiert und (im Falle von Kanten) gerichtet sind, legt das Metamodell fest[15]. Das Metamodell wird durch ein UML-Klassendiagramms definiert. Dabei repräsentiert jede Assoziation einen Kantentyp und jede Klasse einen Knotentyp.

Schema. Das Schema ist die textuelle Beschreibung des Metamodells (in UML). Aus dem Schema können automatisiert Klassen erzeugt werden, welche Knoten- und Kantentypen realisieren. Dabei bekommenen Klassen, die eine Kante implementieren, den Assoziationsnamen zum Klassennamen. Das Schema wird in ein einer `.TG`-Datei, dem JGraLab-Dateiformat [11], festgelegt.

5.2 Anforderungen an das Metamodell

Der Guproansatz soll den Entwickler bei der Softwarewartung unterstützen. Das heisst, dass dieser bei der Arbeit mit den Instanzen des Modells (sprich den TGraphen) auch wie ein Softwareentwickler denken können soll. Im Allgemeinen soll für die Softwarewartung sein Verständnis für das System gefördert werden. Dadurch ergeben sich folgende Anforderungen an das Metamodell:

1. Das Metamodell soll eine feinkörnige Graphenrepräsentation des Quelltextes ermöglichen. Dabei soll sich das Metamodell nicht zu stark an die Syntax von Java anlehnen, aber auch nur so weit davon abstrahieren, dass es einem Entwickler im Sinne des Guproansatzes förderlich ist.

[15] Zusätzlich sind die Elemente geordnet. Dies kann jedoch nicht im Metamodell ausgedrückt werden.

2. Das Metamodell soll eine statische Repräsentation des Quelltextes ermöglichen. Das heisst, dass die TGraphen kein Laufzeitverhalten abbilden sollen.

3. Das Metamodell soll alle Javasprachversionen bis einschließlich 5 abdecken und die Speicherung der Positionen der identifizierten Elemente im Quelltext ermöglichen.[16].

5.3 Metamodell von Hinterwäller

Ein Metamodell für Java 1.4 und die dazu passende Grammatik für Java existierten bereits am Institut für Softwaretechnik. Diese waren im Rahmen der Diplomarbeit [10] von Bodo Hinterwäller entstanden. Es bestand die Hoffnung, dass dieses Metamodell an die Anforderungen dieser Studienarbeit angepasst werden könnte. Das Metamodell basierte auf den Java Development Tools von Eclipse für Java in der Version 1.4. Es musste daher für Java 5 aktualisiert werden. Diese Anpassung wurde vorgenommen, doch im Zuge dieser Überarbeitung wurde offensichtlich, dass eine Verwendung dieses Modells nicht in Frage kommt.

Zuvor war nach einer umfangreichen Onlinerecherche zu möglichen Javaparsern die Wahl auf den Parsergenerator ANTLR in Version 2.7.6 und einer passenden Grammatik für Java 5 gefallen. Der erzeugte Parser sollte das Parsen der Javaklassen für den Faktenextraktor übernehmen. Aufgrund der Basierung auf den Java Development Tools von Eclipse unterschied sich die Grammatik von Hinterwäller zu sehr von jener für ANTLR. Eine Verwendung des Metamodells von Hinterwäller hätte somit die Entwicklung einer neuen Grammatik für ANTLR, basierend auf dem Metamodell von Hinterwäller, bedingt.

Da in dieser Studienarbeit der Fokus auf der Erstellung des Graphen und nicht so sehr auf dem Parsen der Klassen lag, wurde die Entscheidung gefällt, ein eigenes Metamodell zu entwickeln, welches aus der Javagrammatik für ANTLR hergeleitet werden sollte. Als Vorlage für die Herleitung und das zu erreichende Abstraktionsniveau waren die Ergebnisse von Hinterwäller dennoch sehr hilfreich.

Die Erstellung eines eigenen Modells trug schließlich auch sehr zum Verständnis für die Vorgehensweise bei der Umwandlung vom Quelltext zum Graphen bei.

Im nächsten Abschnitt wird das selbstentwickelte Metamodell beschrieben.

5.4 Eigenes Metamodell für Java 5

Das Metamodell wurde aus der Javagrammatik zu ANTLR hergeleitet. Damit sich das Metamodell nicht zu stark an die Syntax von Java anlehnt, sind inhärente Präzedenzen der Syntax

[16]Diese müssen Guprokonform im TGraph mitgespeichert werden, zwecks Nutzung durch den graphenbasierten Sourcecodebrowser

nicht im Metamodell vertreten. Abstrahiert wurde über alle Klammerausdrücke. Diese sind nur implizit durch Strukturen im Metamodell ausgedrückt.

Alle Schlüsselworte aus Java haben ein Pendant als Knoten- oder Kantentyp im Metamodell. Klassen- und Attributsnamen im Modell sind in Anlehnung an Javatermini gewählt. Jeder Knotentyp bezeichnet möglichst genau das Sprachkonstrukt, welches er repräsentiert. Z. B. repräsentiert der Knotentyp `If` die If-Anweisung.

Die syntaktische Reihenfolge kann im Metamodell nicht explizit festgelegt werden, deshalb ist die Reihenfolge im Klassendiagramm graphisch abgebildet. So kann ein Entwickler die vom Programmieren bekannten Strukturen einfach wiedererkennen.

Für die richtige Reihenfolge der Elemente im Graphen sorgt der Faktenextraktor. Er stellt sicher, dass die syntaktische Reihenfolge bei der Erstellung des Graphen miteinfliesst, da die einzelnen Elemente durchnummeriert werden. Ebenso werden mehrere gleichartige Graphelemente (z.B. mehrere Parameter einer Methodendeklaration) in ihrer Ordnung der syntaktischen Reihenfolge entsprechend.

Das Metamodell besteht aus 89 Knoten- und 160 Kantentypen. Diese hier alle zu beschreiben, würde den Rahmen sprengen. Deshalb wird in den nächsten Abschnitt nur ein Überblick gewährt. Die Entwicklung des Metamodells ist in Anhang F detailliert beschrieben.

5.4.1 Grundlegende Knoten- und Kantentypen im Metamodell

Damit ein TGraph mit durch die Guprowerkzeugen weiterverarbeitet werden kann, muss er bestimmte Knoten und Kanten enthalten. Das Metamodell legt die Typen dieser Knoten und Kanten fest, welche im folgenden beschrieben werden.

- `Program` ist der „oberste“ Knotentyp des Metamodells. In einem TGraph existiert jeweils nur ein Knoten dieses Typs; er repräsentiert das geparste Softwareprojekt.
- `TranslationUnit`: eine geparste Datei ist als Knoten diesen Typs im TGraph vertreten. Ist durch eine Kante vom Typ `IsTranslationUnitOf` mit `Program` verbunden.
- `SourceFile`: repräsentiert den Ort einer geparsten Datei und ist durch eine Kante vom Typ `IsPrimarySourceFor` mit `TranslationUnit` verbunden.
- `SourceUsage`: repräsentiert ebenfalls eine geparste Datei und ist durch eine Kante vom Typ `IsSourceUsageOf` mit `TranslationUnit` verbunden.
- `ExternalDeclaration`: jeder Knotentyp, der eine Package-, Import- oder Typdefinition repräsentiert, ist von diesem Typ abgeleitet und durch eine Kante vom Typ `IsExternalDeclarationIn` mit `SourceUsage` verbunden.

Der nächste Abschnitt geht auf Knotentypen ein, die Javasprachkonstrukte repräsentieren.

5.4.2 Knotentypen

Das Metamodell enthält Knotentypen für alle Sprachkonstrukte bis Java Version 5. An dieser Stelle soll nur auf die Hauptgruppen und nicht auf jeden einzelnen der Knotentypen eingegangen werden. Alle im Metamodell definierten Knotentypen basieren auf einer der folgenden Klassen als Basistyp[17]:

- `Type`: Knotentypen, die von diesem abgeleitet sind, repräsentieren Klassen-, Interface-, Enum- und Annotationsdefinitionen sowie Typparameterdeklarationen.
- `TypeSpecification`: Knotentypen, die von diesem abgeleitet sind, repräsentieren Typspezifikationen durch Klassen, primitive Typen, Arrays und Typargumente.
- `Member`: Knotentypen, die von diesem abgeleitet sind, repräsentieren Feld-, Methoden- und Konstruktordeklarationen.
- `Statement`: Knotentypen, die von diesem abgeleitet sind, repräsentieren Anweisungen, wie `if`, `while`, `try`, `catch` usw.
- `Expression`: Knotentypen, die von diesem abgeleitet sind, repräsentieren Ausdrücke, wie `a = b + c`.
- `Annotation`: dieser Knotentyp repräsentiert Annotationen an Package-, Klassen- und Methodendefinitionen.
- `Comment`: Davon abgeleitete Knotentype repräsentieren einzeilige und mehrzeilige Kommentare sowie Javadoc Kommentare.
- `QualifiedName`: repräsentiert qualifizierte Namen im Quelltext, wie `java.util.Vector`.

Die Knotentypen sind in Anhang A zu finden.

5.4.3 Kantentypen

Kantentypen werden im Metamodell durch einfache benannte Assoziationen repräsentiert. Die Leserichtung der Kantennamen ist dabei immer von unten nach oben. Die Kantennamen sind so gewählt, dass durch sie klar ist, welcher Knotentyp an ihrem Ende hängt. Beispielsweise wird durch eine von einem Modifizierer ausgehende Kante `IsModifierOfClass` ersichtlich, dass diese Kante auf eine Klassendefinition zeigt.

Alle Kantentypen, welche einen direkten Bezug zur Syntax von Quelltextes aufweisen, basieren auf der Basistyp `AttributedEdge`. Diese Kanten besitzen Attribute zur Speicherung

[17]Diese sind als abstrakt deklariert und basieren ihrerseits auf dem allgemeinen JGraLab-Knotentyp `Vertex`.

der Positioninformationen, welche sich auf die Entsprechung im Quelltext desjenigen Knotens beziehen, von dem die Kante ausgeht.

Alle Kantentypen, welche einen semantischen Bezug zwischen zwei Knoten abbilden, besitzen diese Attribute nicht und basieren auf dem Kantentyp `Edge` aus JGraLab. Die Kantentypen sind ebenfalls in Anhang A zu finden.

6 Entwurf

In diesem Kapitel wird die Architektur des Javaextraktors beschrieben. Aufgrund der gebotenen Trennung der Belange ist das Programm in verschiedene Klassen und Packages unterteilt, welche im Folgenden beschrieben sind.

6.1 Package javaextractor

Dieses Package enthält die Hauptkomponenten des Javaextraktors. Dies sind folgende Klassen:

JavaExtractor. Diese Klasse enthält die `main()`-Methode, die zum Programmaufruf benötigt wird. Sie beinhaltet die Funktionalität zur Auswertung der übergebenen Kommandozeilenargumente und zur Erstellung der Liste der zu verarbeitenden Dateien. Desweiteren instanziiert diese Klasse den verwendeten Logger und den eigentlichen Graphbuilder.

GraphBuilder. Der GraphBuilder sorgt für den Ablauf des eigentlichen Extraktionsprozesses. Die Klasse instanziiert und initialisiert einen leeren TGraphen, eine leere Symboltabelle, den Lexer, Parser und Resolverklassen (zum Auflösen der Quelltextreferenzen). Dann arbeitet die Klasse die Liste der zu parsenden Dateien ab. Zuletzt sorgt sie für die Speicherung des TGraphen in eine Datei.

SymbolTable. Die Symboltabelle dient zur Zwischenspeicherung von Referenzen und Strukturinformationen des erzeugten TGraph. Dies ist für ein Auflösen durch die Resolverklassen nötig. Ebenso beinhaltet diese Klasse die Funktionalität zum Speichern und Abrufen der Graphelemente, welche im Graph einzigartig sein sollen. Für eine nähere Beschreibung dieser Funktionalität siehe Kapitel 7.3.1.

Utilities. Diese Klasse beinhaltet statische Hilfsmethoden, welche in den anderen Klassen Verwendung finden. Diese dienen dem Füllen der Attibute von Kanten mit konkreten Werten sowie der Fehlerbehandlung.

ExtractionMode. Dieser Enum-Typ repräsentiert die Extraktionsmodi `LAZY`, `EAGER` und `COMPLETE`. Für eine nähere Betrachtung siehe Kapitel 7.3.3.

JavaLexer. Implementiert den Lexer und wird automatisiert von ANTLR aus der Grammatik in `java.g` generiert.

JavaRecognizer. Das Javaparsing ist in dieser Klasse realisiert, welche von ANTLR aus der Grammatik in `java.g` generiert wird. Der Parser erzeugt aus jeder Datei die gültigen Java-Quelltext enthält einen abstrakten Syntaxbaum und nutzt als Lexer `javaextractor.adapters.JavaLexerAdapter`.

JavaTokenTypes. Diese Schnittstelle legt alle durch den Lexer erzeugten Tokenarten fest und wird auch mittels ANTLR aus der Grammatik generiert.

JavaTreeParser. Diese Klassen wird von ANTLR aus der Treewalker-Grammatik in `java.tree.g` erzeugt. Dieser Treewalker dient der Traversierung von abstrakten Syntaxbäumen, die vom Parser erzeugt werden. Er beinhaltet - nachdem in der zugrundeliegenden Grammatik alle dazu nötigen semantischen Aktionen eingefügt wurden - die Funktionalität zum Umsetzen der AST-Struktur in den TGraphen sowie dem Füllen der Symboltabelle mit den nötigen Informationen zum späteren Auflösen der Quelltextreferenzen.

JavaTreeParserTokenTypes. Diese Schnittstelle legt alle Arten der im AST verwendeten Elemente fest und wird ebenfalls durch ANTLR aus der Treewalker-Grammatik erzeugt.

6.2 Package javaextractor.adapters

Dieses Package enthält all jene Klassen, welche die Anpassungen kapseln, die für die Funktionalität der Positionsabfrage von Sprachelementen aus dem Quelltext notwendig sind. Der Implementierung der Klassen liegt das Adapter-Pattern (siehe [7] auf den Seiten 139ff) zugrunde.

CommonTokenAdapter. Diese Klasse implementiert ein Token, welches seinen Offset kennt und bietet Methoden, die zum Speichern und Abrufen der Positionsinformationen des Tokens dienen.

LexerSharedInputStateAdapter. Diese Klasse dient dem Lexer zur Verwaltung des Eingabestroms der aktuell geparsten Datei. Es kennt die aktuelle Position im Eingabestrom in Form eines Offsets und kann alle Positionsinformationen in ein Token vom Typ `CommonTokenAdapter` schreiben.

JavaExtractor
+ main(String[]) : void

uses

«enumeration»
ExtractionMode
«enum»
LAZY
EAGER
COMPLETE

GraphBuilder
+ finalizeGraph(ExtractionMode) : void
+ GraphBuilder(String, Logger)
+ parseFiles(Vector< String >) : void
+ saveGraphToDisk(String) : void

uses

antlr::LLkParser

JavaRecognizer

uses

antlr::TreeParser

CharScannerAdapter
JavaLexer

javaextractor.adapters::JavaLexerAdapter
+ JavaLexerAdapter(LexerSharedInputState)
makeToken(int) : Token
+ newline() : void
+ tab() : void

JavaTreeParser

fills

uses

Utilities
+ addToFullyQualifiedName(Vertex, AST) : void
+ copyEdgeAttributes(AttributedEdge, AttributedEdge) : void
+ fillEdgeAttributes(CommonASTAdapter, AttributedEdge) : void
+ fillEdgeAttributesFromAST(AttributedEdge, AST) : void
+ fillEdgeAttributesFromASTDifference(AttributedEdge, AST, AST) : void
+ fillEdgeAttributesWithGivenValue(AttributedEdge, int) : void
+ stackTraceToString(Exception) : String

SymbolTable
+ addIntegerConstant(String, IntegerConstant) : void
+ addJavaPackage(String, JavaPackage) : void
+ addModifier(Modifiers, Modifier) : void
+ addTypeDefinition(String, Type) : void
+ getGraph() : Java5
+ getIntegerConstant(String) : IntegerConstant
+ getJavaPackage(String) : JavaPackage
+ getModifier(Modifiers) : Modifier
+ getNull() : Null
+ getTypeDefinition(String) : Type
+ hasEmptyStatement() : boolean
+ hasIntegerConstant(String) : boolean
+ hasJavaPackage(String) : boolean
+ hasModifier(Modifiers) : boolean
+ hasNull() : boolean
+ hasTypeDefinition(String) : boolean
+ nextFile() : void
+ reset() : void
+ setEmptyStatement(EmptyStatement) : void
+ setNull(Null) : void
+ SymbolTable(Java5)

Abbildung 3: Klassendiagramm des Package `javaextractor`

CharScannerAdapter. Verwendet den `LexerSharedInputStateAdapter`, um die Token mit Positionsinformationen zu versorgen.

JavaLexerAdapter. Erzeugt Token vom Typ `CommonTokenAdapter`, welche die benötigten Positionsinformationen tragen.

CommonASTAdapter. Im Gegensatz zu ihrer Oberklasse hält diese Klasse eine Referenz auf ihr zugrundeliegendes Token. Das Token ist ein `CommonTokenAdapter`, welches seinen Offset kennt. Die Positionsinformationen können mit Methoden dieser Klasse abgerufen werden.

6.3 Package javaextractor.comments

Die Klassen dieses Packages repräsentieren Kommentare, die vom Lexer im Quelltext gefunden wurden. Gefundene Kommetare können somit in den TGraph, an Parser und Treewalker vorbei, aufgenommen werden. Der Suffix `Class` im Namen einer Klasse dient zur Unterscheidung vom Knotentyp im TGraph.

CommentClass. Diese abstrakte Klasse definiert die gemeinsame Funktionalität aller Klassen, die Kommentare repräsentieren. Alle Klassen dieses Packages erben von ihr und implementieren jeweils die Funktionalität zum Erzeugen ihres Pendants im TGraph..

SingleLineCommentClass. Repräsentiert einen einzeiligen Kommentar im Quelltext.

MultiLineCommentClass. Repräsentiert einen mehrzeiligen Kommentar im Quelltext, der kein Javadoc-Kommentar ist.

JavaDocCommentClass. Repräsentiert einen Javadoc-Kommentar im Quelltext.

Abbildung 4: Klassendiagramm des Package `javaextractor.comments`

6.4 Package javaextractor.factories

Dieses Package enthält Klassen, die Funktionalität zum Erstellen von Teilstrukturen im TGraph aus Elementen des ASTs implementieren. Diese Klassen werden in `JavaTreeParser` verwendet, so dass der Treewalker selbst kompakt bleibt und dort mehrfach benötigte Funktionen nur einmal implementiert werden müssen. Die Implementierung der Klassen ist an die Patterns Abstract Factory (siehe [7] auf den Seiten 87ff) und Factory Method (siehe [7] auf den Seiten 107ff) angelehnt.

SubgraphFactory. Diese abstrakte Klasse stellt die in allen weiteren Factory-Klassen benötigte Funktionalität - zum Halten einer Referenz auf den TGraph und die Symboltabelle - bereit. Alle Klassen dieses Packages erben von ihr.

Konkrete Factories. Folgende Liste führt auf für welche Quellcodeelemente die jeweilige Klasse die entsprechenden Teilstrukturen im TGraph generiert. Die Klassen sorgen auch dafür, dass semantisch identische Knoten nur einmal im TGraph vorkommen, sofern dies zu Zeitpunkt des Aufbau des TGraphen eindeutig entscheidbar ist.

- `AnnotationFactory`: Annotationen und Annotationsfelder

- `ConstantFactory`: `null`-, `boolean`-, `float`-, `double`-, `int`-, `long`-, `char`- und `String`-Literale
- `ExpressionFactory`: Konstruktor- und Methodenaufrufe, Typumwandlungen, Präfix-, Infix- und Postfix-Operator-Ausdrücke sowie Array- und Objekt-Instanziierung
- `FieldFactory`: Deklarationen und Zugriffe auf Felder, Variablen und Enum-Werten
- `IdentifierFactory`: einfache Bezeichner
- `ImportFactory`: Importdefinitionen
- `MemberFactory`: Konstruktor- und Methodendefinitionen sowie Methodendeklarationen
- `ModifierFactory`: Modifizierer
- `PackageFactory`: Packagedefinitionen
- `QualifiedNameFactory`: qualifizierte Bezeichner
- `StatementFactory`: Labels sowie `If`-, `For`-, `While`-, `Do..While`-, `Break`-, `Continue`-, `Return`-, `Switch`-, `Try`-, `Throw`-, `Synchronize`- und `Assert`-Anweisungen
- `TypeDefinitionFactory`: Klassen-, Schnittstellen-, Enum- und Annotationsdefinitionen
- `TypeParameterFactory`: Typparameter und -argumente
- `TypeSpecificationFactory`: Typspezifikationen

6.5 Package javaextractor.resolvers

Dieses Package enthält Klassen, die dem Auflösen der Quelltextreferenzen dienen.

Resolver Diese Klasse ist abstrakt und beinhaltet die für alle Resolverklassen gemeinsame Funktionalität. Alle weiteren Klassen im Package, mit Ausnahme von `ResolverUtilities`, erben von ihr.

LocalResolver Kann Quelltextreferenzen auf lokal definierte Elemente auflösen.

TypeSpecificationResolver Löst Typspezifikationen auf, die nicht bereits durch den `LocalResolver` aufgelöst werden konnten.

FieldResolver Implementiert Methoden zum Auflösen von Zugriffen auf Variablen, Felder und Enum-Konstanten innerhalb des TGraphen.

MethodResolver Stellt Methoden bereit zum Auflösen von Methoden- und Konstruktoraufrufen innerhalb des TGraphen.

ResolverUtilities Beinhaltet statische Hilfsmethoden zur Verwendung durch die Resolver-Klassen.

6.6 Package javaextractor.schema

In diesem Package sind Schnittstellen enthalten, welche von den Knoten- und Kantentypen der erzeugten TGraphen implementiert werden. Alle Quelltexte in diesem Package wurden durch das JGraLab-Werkzeugs `TgSchema2Java` aus dem Schema (siehe Anhang B) erzeugt. Aufgrund der Menge der Typen soll hier jedoch nur auf generelle Eigenschaften eingegangen werden.

Jede dieser Schnittstellen legt die Eigenschaften eines Knoten- oder Kantentyps fest. Für jeden gleichnamigen Knoten- und Kantentyp aus dem Metamodell existiert eine Schnittstelle. Statt Klassen werden Schnittstellen erzeugt, da im Metamodell Mehrfachvererbung erlaubt ist und Java dies nur in Schnittstellen erlaubt.

6.7 Package javaextractor.schema.impl

In diesem Package sind Kanten- und Knotenklassen enthalten, die in den TGraphen vorkommen, welche vom Javaextraktor erzeugt werden. Auch diese wurden automatisch von `TgSchema2Java` aus dem Schema (siehe Anhang B) generiert. Für jeden Knoten- und Kantentyp aus dem Metamodell existiert hier eine gleichnamige Klasse mit dem Suffix „Impl“, welche ihre Schnittstelle sowie die Schnittstellen ihrer „Oberklassen“ aus `javaextractor.schema` implementiert. Jede Kantenklasse implementiert `de.uni_koblenz.jgralab.Edge` und jede Knotenklasse `de.uni_koblenz.jgralab.Vertex`.

Zusätzlich dazu wurde für jeden Kantentyp eine „Reversed“-Klasse generiert. Dabei handelt es sich um die jeweilige Kante in umgekehrter Richtung, was eine effizientere Traversierung der generierten TGraphen ermöglicht.

Resolver
symbolTable: SymbolTable
+ attach(Type, QualifiedType) : void
+ attach(String, QualifiedType) : void
+ replaceByResolvedOne(String, QualifiedType) : void

LocalResolver
+ LocalResolver(SymbolTable) : void
+ resolve() : boolean

TypeSpecificationResolver
+ resolveTypeSpecifications(ExtractionMode) : boolean
+ TypeSpecificationResolver(SymbolTable)

FieldResolver
+ FieldResolver(SymbolTable)
+ resolveFields(ExtractionMode) : boolean
resolveSingleField(ExtractionMode, FieldAccess) : boolean
+ setMethodResolver(MethodResolver) : void

uses 1 1

MethodResolver
+ MethodResolver(SymbolTable)
+ resolveMethods(ExtractionMode) : boolean
resolveSingleMethod(ExtractionMode, MethodInvocation) : void
+ setFieldResolver(FieldResolver) : void

uses 1 1

uses 1 1

uses 1

ResolverUtilities
+ accessedDimensionsMatchDeclaredDimensions(FieldAccess, ArrayType) : boolean
+ createAndAttachEnumConstants(Object[ ], Type, ExtractionMode, SymbolTable) : void
+ createAndAttachFields(Field[ ], Type, ExtractionMode, SymbolTable) : void
+ createAndAttachModifiers(int, Vertex, SymbolTable) : void
createConstructorDefinition(Constructor, Type, ExtractionMode, SymbolTable) : ConstructorDefinition
createEnumConstant(Object, Type, SymbolTable) : EnumConstant
createFieldDeclaration(Field, Type, ExtractionMode, SymbolTable) : void
createMethodDeclarationOrDefinition(Method, Type, ExtractionMode, SymbolTable) : MethodDeclaration
+ createType(Class, ExtractionMode, SymbolTable) : Type
+ createTypeSpecification(Class<?>, SymbolTable, ExtractionMode) : TypeSpecification
+ createTypeUsingReflection(String, ExtractionMode, SymbolTable) : boolean
+ getCastedTypeFromClassCast(ClassCast) : Type
+ getEnclosingClassFromScope(Vertex, SymbolTable) : ClassDefinition
+ getEnclosingTypeFromScope(Vertex, SymbolTable) : Type
+ getFullyQualifiedName(ImportDefinition) : String
+ getImportsFromSupremeType(Type) : ArrayList< ImportDefinition >
+ getOrCreateTypeBlock(Type, SymbolTable) : Block
+ getPackageNameFromSupremeType() : String
+ getReturnTypeFromMethodInvocation(MethodInvocation, ExtractionMode, MethodResolver) : Type
+ getSuperClass(ClassDefinition) : ClassDefinition
+ getSupremeTypeFromScope(Vertex, SymbolTable) : Type
+ getTypeFromFieldDeclaration(FieldDeclaration, FieldAccess) : Type
+ getTypeFromObjectCreation(ObjectCreation) : Type

Abbildung 5: Klassendiagramm des Package `javaextractor.resolvers`

7 Funktionsweise des Javaextraktors

Der Ablauf des Extraktionsprozess ist in mehrere voneinander abgegrenzte Arbeitsschritte aufgeteilt. Einen Überblick verschafft das Aktivitätsdiagramm in Abbildung 6. Die Arbeitsschritte werden in folgender Reihenfolge ausgeführt:

1. Die übergebenen Kommandozeilenparameter (eine genaue Beschreibung folgt in Abschnitt 8.2) werden interpretiert und die entsprechenden Einstellungen gesetzt.
2. Aus allen Dateien und Verzeichnissen, welche beim Aufruf übergeben wurden, wird eine Liste der enthaltenen `.java`-Dateien erstellt. Im Falle von Verzeichnissen werden auch alle enthaltenen Unterverzeichnisse rekursiv mit einbezogen. Angegebene, aber nicht existente Elemente werden dabei ignoriert, ebenso alle Dateien, welche nicht auf die Erweiterung `.java` enden.
3. Es wird ein leerer, dem verwendeten Schema entsprechender TGraph erzeugt. Darüber hinaus werden einige grundlegende Knoten und Kanten im Graph erzeugt und initialisiert. Zusätzlich wird eine leere Symboltabelle erstellt.
4. Für jede Quelltextdatei in der Liste werden die folgenden Schritte abgearbeitet:
   - Der Parser erzeugt einen AST aus der Datei.
   - Im TGraph werden die entsprechenden Verwaltungsinformationen für die Datei erzeugt.
   - Dem Treewalker wird der AST, der TGraph, die Symboltabelle sowie einige weitere Objektreferenzen übergeben.

     Der Treewalker traversiert den AST und generiert dabei die entsprechenden Elemente im TGraph in einer dem Metamodell entsprechenden Struktur. Ebenso werden weitere Objektreferenzen gesammelt und zur Symboltabelle hinzugefügt.
   - Innerhalb der erzeugten Teilstruktur des TGraph werden lokale Typreferenzen und Variablenzugriffe aufgelöst und die entsprechenden Kanten erzeugt. Dabei semantisch identische, mehrfach vorkommende Knoten werden zu einem einzigen Knoten zusammengeführt.
5. Über die gesamte Struktur des Graphen werden Typreferenzen, Variablenzugriffe und Methodenaufrufe aufgelöst, entsprechende Kanten erzeugt und wiederum semantisch identische, mehrfach vorkommende Knoten verschmolzen.

   Sind Elemente auch dabei nicht auflösbar, so wird, in Abhängigkeit vom eingestellten Modus, mittels der `Reflection`-Funktionalität von Java eine Auflösung versucht.
6. Der TGraph wird in einer Datei abgespeichert, dabei wird der übergebene Pfad und Dateiname verwendet. Existiert die Datei bereits, so wird diese überschrieben.

Abbildung 6: Aktivitätsdiagramm Ablauf des Extraktionsprozesses

Bei diesem Ablauf ist zu beachten, dass bei einem Programmaufruf für alle dabei analysierten Quelltexte nur ein gemeinsamer Graph erzeugt wird. Eine genaue Beschreibung zu einzelnen Punkten folgt in den nächsten Abschnitten.

7.1 Erzeugung des TGraphen im Treewalker

Die Umsetzung der Struktur des vom Parser für jede Quelltextdatei erzeugten AST zum TGraph geschieht im Treewalker. Grundsätzlich wäre es möglich (und hinsichtlich Ausführungsgeschwindigkeit und Speicherverbrauch etwas effizienter) den TGraph bereits während des Parsings mit aufzubauen. Die Grammatik des Treewalkers ist aber erheblich kompakter und lesbarer als die des Parsers. Dies vereinfacht die Implementierung wesentlich. Gerade zur Vermeidung und Suche von Fehlern ist eine kompakte Grammatik von Vorteil. Darüberhinaus bietet die Arbeitsweise des Treewalkers eine einfache Möglichkeit zum Erzeugen der TGraph-Elemente, da die Struktur seiner Regeln derjenigen ähnelt, die der generierte Graph gemäß dem Metamodell besitzen soll.

Zur Erzeugung des TGraphen ist es nicht nötig, den Treewalker selbst anzupassen, sondern es genügt, entsprechenden Code als semantische Aktionen in seine zugrundeliegende Grammatik einzufügen. Ein aus der modifizierten Grammatik erzeugter Treewalker beinhaltet dann die gewünschte Funktionalität.

Zunächst müssen dem Treewalker Referenzen mitgegeben werden, damit dieser die zu erzeugenden Graphelemente an die richtigen Stellen im Graphen verbinden kann. Diese werden mittels Setter-Methoden vom Javaextraktor gesetzt, bevor der AST traversiert wird. Dabei handelt es sich um die folgenden Elemente:

- `programGraph`: Der TGraph, der erweitert werden soll.
- `programVertex`: Der Wurzelknoten des Graphen.
- `sourceUsageVertex`: Der Knoten im Graphen, in dem der Pfad der aktuellen Quelltextdatei gespeichert ist.
- `translationUnitVertex`: Der Knoten im Graphen, der als Wurzelknoten für alle Elemente aus der aktuellen Quelldatei dient.
- `symbolTable`: Die zum TGraph gehörige Symboltabelle.

Die eigentliche Generierung einer Teilstruktur des TGraphen im Treewalker entspricht in den meisten Fällen dem folgenden Prinzip:

1. Am Beginn einer Regel wird ein Knoten der zur Regel äquivalenten Knotenklasse im TGraph erzeugt.

2. Mit diesem Knoten wird nach jeder referenzierten Unterregel der von der Unterregel zurückgegebene Knoten durch Erzeugen einer Kante der entsprechenden Kantenklasse verbunden.

3. Am Ende der Regel wird der in Punkt 1 erzeugte Knoten zurückgegeben.

Beim Erzeugen des Treewalkers aus der Grammatik generiert ANTLR aus jeder Regel eine eigene Methode. Diese besitzen als Rückgabewert immer den Knoten des AST (vom Typ `AST`; dieser ist in ANTLR definiert), der von der Regel durchlaufen wird. Deswegen wird der erstellte TGraph-Knoten am Ende der Regel in einer globalen Variable `currentVertex` - vom in JGraLab definierten Typ `Vertex` - zwischengespeichert.

Eine Ausnahme zum o.a. Prinzip bildet die Listenregel. Dabei handelt es sich um eine Regel, die zwar selbst wiederum mehrere Regeln referenziert, jedoch zu ihr selbst keine Entsprechung als Knoten im TGraphen erzeugt. Zum Beispiel referenziert die Regel einer Methodendeklaration die Regel `parameters`, welche beliebig oft die Regel `parameterDeclaration` referenziert. Gemäß dem Schema muss aber direkt eine Kante zwischen jeder Parameterdeklaration und der Methodendeklaration erzeugt werden. In solchen Fällen übergibt die referenzierende Regel den eigenen Knoten (z.B. vom Typ `MethodDeclaration`) an die Listenregel (ebenfalls über die globale Variable `currentVertex`) und diese erzeugt die entsprechenden Kanten (z.B. vom Typ `IsParameterOfMethod`).

Mit der beschriebenen Vorgehensweise funktioniert die komplette Erzeugung der grundlegenden TGraph-Strukturen, da die entsprechende Funktionalität durchgängig in allen Regeln implementiert ist.

7.2 Sammeln und Berechnen der Positionsinformationen

Durch die bereits in Abschnitt 4.3 beschriebenen Anpassungen ist es möglich, im Treewalker Positionsinformationen zu den einzelnen Elementen des ASTs abzufragen und zu verarbeiten. Somit konnte das Berechnen und Speichern der jeweiligen Attribute in den TGraph implementiert werden. Die implementierte Funktionalität ist in den relevanten Treewalker-Regeln enthalten und ähnelt im Prinzip der Generierung der einzelnen Elemente des TGraphen:

1. Nach jeder referenzierten Unterregel wird das zurückgegebene Anfangs- und End-AST-Element abgerufen und daraus die Positionsinformationen sowie die Gesamtlänge berechnet. Diese werden in die Kante eingetragen, welche den von der Regel zurückgegebenen Knoten mit anbindet (bei einer Listenregel nicht).

2. Am Ende der Regel wird das eigene Anfangs- und End-AST-Element zurückgegeben. Dabei handelt es sich entweder um das erste / letzte Element aus den Unterregeln oder um eventuell in der Regel direkt vorhandene AST-Elemente.

Für die Ermittlung werden zwei globale Variablen verwendet (`currentBeginAST` und `currentEndAST`). Mit Ausnahme der Kanten, unter denen sich keine zusammengesetzte Struktur befindet[18], findet die beschriebene Funktionalität im Treewalker durchgehend Verwendung.

7.3 Auflösen der Quelltextreferenzen

Nach dem Durchlaufen eines AST ist dessen Struktur im wesentlichen in einen TGraph umgesetzt. Der TGraph soll die Syntax der Quelltexte jedoch in Form eines ASG repräsentieren. Im Gegensatz zum AST sind bestimmte Knoten einzigartig und Schleifen erlaubt. Grundsätzlich soll im Graphen jeder Knoten, der eine Typspezifikation, einen Methoden-/Konstruktoraufruf oder einen Feld-/Variablenzugriff repräsentiert über eine Kante mit dem Knoten verbunden sein, welcher die entsprechende Definition bzw. Deklaration repräsentiert. Das Auflösen von Referenzen umfasst neben dem Ermitteln der spezifizierten Typen, Typparameter, Variablen, Label und Methoden auch eine entsprechende Anpassung des Graphen.

7.3.1 Genereller Ablauf des Auflösevorgangs

Zunächst werden während der Ausführung des Treewalkers jene Knoten in der Symboltabelle gesammelt, die für das Auflösen der Referenzen von Bedeutung sind. Dies sind Knoten, die eines der folgenden Elemente im Quelltext repräsentieren:

- Typdefinitionen
- Typspezifikationen
- Parameterdeklarationen
- Variablendeklarationen
- Variablenzugriffe (dies beinhaltet auch Zugriffe auf Felder, Parameter und Enum-Werte)
- Methodendeklarationen und -definitionen
- Konstruktordefinitionen
- Methodenaufrufe (dies beinhaltet auch Konstruktoraufrufe)

Das Auflösen aller Quelltextreferenzen geschieht dann in zwei Schritten. Im ersten Schritt wird nach dem Parsen jeder Datei versucht Referenzen lokal aufzulösen, die in dieser Datei gefunden wurden; d. h. sie werden mit Definitionen abgeglichen, die nur lokal in dieser Datei sichtbar sind (z. B. Definitionen von lokalen Variablen, inneren Klassen usw.).

[18] Dies ist nur bei `Identifier` der Fall.

Quelltextreferenzen, die auf diese Weise nicht aufgelöst werden können, werden im zweiten Schritt erneut behandelt. Nach dem Parsen aller Quelltexte sind alle o. a. Definitionen im Graph vertreten und in der Symboltabelle gespeichert. Diesmal werden die Quelltextreferenzen mit global sichtbaren Definitionen abgeglichen. Sollte das Auflösen dabei ebenfalls nicht möglich sein, wird (abhängig vom Modus des Extraktors) auch eine Auflösung per Introspektion[19] versucht. Dabei können nur Elemente gefunden werden, die über den `CLASSPATH` erreichbar sind.

7.3.2 Anpassung des Graphen während des Auflösevorgangs

Beim Auflösen wird eine Quelltextreferenz über eine Kante mit ihrer Definition verbunden. Konkret werden dazu folgende Kanten erzeugt:

- `IsTypeDefinitionOf` von `Type`- zu `TypeSpecification`-Knoten.
- `IsInvokedMethod` von `MethodDeclaration`-, `MethodDefinition`- bzw. `ConstructorDefinition`- zu `MethodInvocation`-Knoten.
- `IsDeclarationOfAccessedField` von `VariableDeclaration`-, `ParameterDeclaration`- bzw. `EnumConstant`- zu `FieldAccess`-Knoten.

Da es sich dabei um semantische Kanten handelt, sind diese nicht mit Positionsinformationen attributiert.

Beim Auflösen einer Typspezifikation muss ggf. der Name des spezifizierten Typs, der in einem Knoten mitgespeichert wird, angepasst werden. Dies ist der Fall, wenn eine Typspezifikation nicht vollqualifiziert im Quelltext vorliegt. Wird beispielsweise eine Variable `v` per `Vector v;` deklariert, so ist die Typspezifikation `Vector` nicht vollqualifiziert. Im Graphen existiert somit ein Knoten vom Typ `QualifiedType`, der in seinem Attribut `fullyQualifiedName` den String `Vector` speichert. Die Auflösung ergibt, dass die Klasse `java.util.Vector` gemeint ist. Der vollqualifizierte Name `java.util.Vector` wird dann in den Knoten eingetragen.

Im Graphen sollen Knoten, die eine Variable, einen Typ oder eine Methode repräsentieren, nur einmal vorkommen. In einem Quelltext werden solche Elemente definiert, auf die dann über deren Namen zugegriffen wird. Im Quelltext kann somit der Name mehrfach vorkommen. Im Graphen hingegen soll dessen Repräsentation einzigartig sein. Für jedes Vorkommen im Quelltext soll im Graphen aber eine Kante erzeugt werden. Um dies sicherzustellen wird beim Auflösen einer Referenz überprüft, ob diese bereits zuvor aufgelöst wurde und somit eine korrekte Graphenrepräsentation schon existiert. Ist dies der Fall, wird diese wiederverwendet.

Zum Beispiel wird für jede Verwendung einer Klasse im Quelltext ein Knoten vom Typ `QualifiedType` erzeugt. Wird diese Typspezifikation zum ersten Mal aufgelöst, so wird

[19] In Java über Reflexion.

dieser Knoten in der Symboltabelle als aufgelöst geführt und kann wiederverwendet werden. Bei der nächsten Auflösung der gleichen Klasse wird dann auch der selbe Knoten aus der Symboltabelle verwendet und der überflüssige Knoten gelöscht. Zuvor werden jedoch alle Kanten des zu löschenden Knotens umgehangen.

Im Falle von Methodenaufrufen und Variablenzugriffen wird sichergestellt, dass die verbundenen Identifier-Knoten einzigartig sind. Die jeweiligen `MethodInvocation`- und `FieldAccess`-Knoten bleiben unverändert, da bei jedem Methodenaufruf andere Argumente übergeben werden können oder bei jedem Zugriff auf eine Array-Variable auf ein anderes Element des Arrays[20] zugegriffen werden kann.

7.3.3 Auflösemodi

Der Javaextraktor kann wahlweise mit drei verschiedenen Auflösemodi benutzt werden. Sie unterscheiden sich in der Tiefe des Auflösens externer[21] Typen, ihrer Felder und Methoden.

Externe Typen sind daran erkennbar, dass das Attribut `external` auf `true` gesetzt ist. Knoten, die Felder, Methoden und Konstruktoren repräsentieren, besitzen dieses Attribut nicht. Für sie gilt der Wert des Attributs des Typdefinitionsknoten an dem sie hängen. Der Knoten der Typdefinition ist dabei über zwei Kanten erreichbar: über `IsMemberOf` zum `Block` der Typdefinition und von dort über `IsBlockOf` zur Typdefinition selbst.

LAZY. Dies ist der Standardmodus. In diesem Modus werden Referenzen nur im Rahmen der zu parsenden Quelltexte aufgelöst. Eine Quelltextreferenz kann nur aufgelöst werden, wenn die Klasse, die das referenzierte Element definiert, ebenfalls vom Javaextraktor im selben Arbeitsgang geparst wird; d. h. dass z. B. Typen aus der Java-API nur aufgelöst werden könnten, wenn sie auch vom Javaextraktor geparst werden würden.

Dieser Modus sollte benutzt werden, wenn Referenzen zu externen Klassen bei der Analyse des Softwareprojekts nicht von Interesse sind. Zu beachten ist allerdings auch, dass in diesem Modus auch keinerlei interne Methoden aufgelöst werden können, welche einen Parameter eines externen Typs besitzen (was schon auf einen einfachen `String` zutrifft).

EAGER. In diesem Modus werden Referenzen im Rahmen der zu parsenden Quelltexte und dem `CLASSPATH` aufgelöst. Referenzen, die nicht im LAZY-Modus aufgelöst werden können, werden im EAGER-Modus per Reflection im `CLASSPATH` gesucht. Definitionen, die über den `CLASSPATH` erreichbar sind, sind nicht im Graphen repräsentiert (da ihre Klassen nicht geparst wurden). Dies muss somit noch nachgeholt werden. Ist ein Auflösen erfolgreich,

[20] Es kann auch auf das gesamte Array oder eine der Dimensionen eines mehrdimensionalen Arrays zugegriffen werden.

[21] nicht in den Quelltexten der geparsten Dateien definiert.

so wird ein Knoten zur Repräsentation der Definition erzeugt, um den Knoten, der die Referenz repräsentiert, per entsprechender Kante zu verbinden.

Mit diesem Modus wird ein Graph erzeugt, in dem ersichtlich ist, ob eine Referenz im Rahmen der geparsten Klassen bleibt oder externe Elemente, wie z. B. Java-API berührt. Erkennbar ist dies an den Elementen, die eine Definition im Graphen repräsentieren. Besteht die Repräsentation nur aus einem einzigen Knoten, so konnte die entsprechende Referenz nur per Reflection aufgelöst werden. Besteht die Repräsentation aus mehreren Elementen (z. B. ganze Klassensignatur) so konnte die entsprechende Referenz im Rahmen der geparsten Quelltexte aufgelöst werden. Referenzen, die überhaupt nicht aufgelöst werden konnten, sind im Graphen mit keiner Definition verbunden.

Dieser Modus sollte benutzt werden, wenn Referenzen zu externen Klassen bei der Analyse des Softwareprojekts von Belang sind, jedoch darüberhinaus keine Analyse nötig ist. Außerdem kann festgestellt werden, ob im geparsten Softwareprojekt Klassen fehlen. In diesem Modus können nur solche Felder und Methoden nicht aufgelöst werden, die in einer nicht direkt referenzierten Oberklasse einer direkt im Quelltext referenzierten Klasse definiert sind.

COMPLETE. In diesem Modus verhält sich der Javaextraktor zunächst wie im Modus `EAGER`. Es werden jedoch beim Auflösen per Reflection immer komplette Graphrepräsentationen erzeugt. Für jede hinzukommende Referenz wird eine komplette Repräsentation der Klasse im Graphen erzeugt, außer wenn diese bereits vorhanden ist. Zusätzlich müssen die dadurch hinzukommenden Referenzen selbst wieder aufgelöst werden. Da dieser Vorgang rekursiv abläuft und nicht vorhersehbar ist, wie viele Referenzen in den externen Klassen aufgelöst werden müssen, kann dies u. U. sehr lange dauern.

Dieser Modus sollte benutzt werden, wenn externe Klassen bei der Analyse des Softwareprojekts von Interesse sind.

7.4 Aufwandsbetrachtung

Die Dauer des Extraktionsprozesses und die Größe des extrahierten Graphen hängen von der Größe des zu parsenden Softwareprojektes ab. Dazu wurden verschiedene Softwareprojekte geparst und dabei die Anzahl der Quelltextzeilen gezählt sowie die benötigte Ausführungsdauer und die Größe des Graphen festgehalten. Geparst wurden die Quelltexte von ANTLR, JGraLab und Packages des Javaextraktors, die das Schema realisieren (`javaextractor.schema` und nochmals separat `javaextractor.schema.impl`). Jedes Softwareprojekt wurde in allen drei Modi jeweils zwei Mal geparst und der durchschnittliche Zeitaufwand ermittelt. Tabelle 3 listet die gemessenen Werte für die Extraktionsmodi `LAZY`, `EAGER` und `COMPLETE` auf[22].

[22] Alle Messungen wurden auf einem AMD Athlon64 3200+ mit 2GB RAM unter Windows XP vorgenommen.

An den gemessenen Werten ist zu erkennen, dass der Aufwand generell linear mit der Zahl der Codezeilen ansteigt. Erwartungsgemäß ist die Zeitaufwand im Modus `COMPLETE` am größten und im Modus `LAZY` am niedrigsten. Teilweise ist der Zeitaufwand jedoch in `EAGER` am größten, da in diesem Modus ein und dieselbe Klasse ggf. mehrfach per Introspektion untersucht werden muss. Im Modus `EAGER` ist jede externe Klassendefinition nach dem Auflösen der Typspezifikationen per Reflexion jeweils nur mit einem Knoten im TGraph vertreten. Für die Auflösung der Methodenaufrufe werden mehr Informationen benötigt, so dass die betroffene Klasse nochmals per Reflexion untersucht wird.

Kann eine Quelltextreferenz aufgelöst werden, wird im Graph ein Knoten gelöscht und dafür eine Kante erzeugt. So ist im Modus `LAZY` die Anzahl der Knoten höher und die Anzahl der Kanten niedriger, als in den Modi `EAGER` und `COMPLETE`, da weniger Quelltextreferenzen aufgelöst werden können.

Softwareprojekt	Extraktormodus	Anzahl der Klassen	Anzahl der Quelltextzeilen	Dauer Parsing & Graphaufbau in s	Dauer Resolving in s	Dauer Graph speichern in s	Gesamtdauer in s	Anzahl der Knoten im Graph	Anzahl der Kanten im Graph
antlr	lazy	216	55.725	14,77	6,62	1,16	22,55	133.557	241.298
antlr	eager	216	55.725	14,80	10,02	1,23	26,05	132.410	259.297
antlr	complete	216	55.725	14,67	11,56	1,70	27,93	178.624	383.131
de.uni_koblenz.jgralab	lazy	370	68.888	13,93	3,67	1,06	18,66	116.491	190.181
de.uni_koblenz.jgralab	eager	370	68.888	13,42	7,96	1,13	22,51	118.616	211.990
de.uni_koblenz.jgralab	complete	370	68.888	12,75	7,72	1,23	21,70	130.706	243.142
javaextractor.schema.impl	lazy	369	141.584	65,13	21,62	4,86	91,81	663.345	807.483
javaextractor.schema.impl	eager	369	141.584	65,14	393,36	4,12	463,42	483.898	861.269
javaextractor.schema.impl	complete	369	141.584	65,12	358,83	4,86	428,81	489.732	872.198
javaextractor.schema	lazy	626	232.947	79,27	135,34	5,07	219,68	685.715	971.397
javaextractor.schema	eager	626	232.947	81,24	465,43	4,67	551,34	565.984	987.669
javaextractor.schema	complete	626	232.947	84,78	436,87	4,79	526,44	576.269	1.008.979

Tabelle 3: Ergebnisse der Messungen in den verschiedenen Extraktormodi

8 Verwendung des Javaextraktors

8.1 Erzeugung

Da der Extraktor teils als Quelltext und teils als ANTLR-Grammatik vorliegt, müssen aus diesen einmalig die entsprechenden Programme erzeugt werden, bevor eine Verwendung stattfinden kann. Folgende Schritte müssen dazu durchgeführt werden:

1. Im Folgenden wird die Verzeichnisstruktur verwendet, die aus dem SVN-Repository des Gupro-Projektes (siehe [19]) stammt.

   In ein lokales Arbeitsverzeichnis `localrepository` werden zunächst die drei Verzeichnisse

   - `localrepository/jgralab`
   - `localrepository/common`
   - `localrepository/javaextractor`

   aus dem Online-Repository abgelegt (mittels `svn checkout`). Abbildung 7 zeigt die (auf wesentliche Verzeichnisse reduzierte) Verzeichnisstruktur. Alle weiteren Schritte werden ausgehend vom Verzeichnis `localrepository/javaextractor/src` durchgeführt.

2. Danach müssen folgende Verzeichnisse und Dateien in den `CLASSPATH` aufgenommen werden:

   - Das aktuelle Verzeichnis:
     `"."`
   - Das Verzeichnis mit den Quellen von JGraLab:
     `"../../jgralab/src"`
   - Das von JGraLab teilweise verwendete GetOPT:
     `"../../common/lib/getopt/java-getopt-1.0.13.jar"`
   - Das vom Extraktor verwendete ANTLR:
     `"../../common/lib/antlr/org.antlr_2.7.6.jar"`

   Am einfachsten geschieht dies durch Setzen einer entsprechenden Umgebungsvariable. Die Alternative mittels Übergabe des Java-Kommandozeilenparameters `-cp` bei Kompilierung und Ausführung ist auch möglich, hier wird jedoch ersteres angenommen.

3. Anschließend müssen Lexer, Parser und Treewalker aus ihren Grammatiken durch ANTLR im Verzeichnis `localrepository/javaextractor/src/javaextractor` erzeugt werden. Die Erzeugung geschieht durch die folgenden zwei Aufrufe:

```
java antlr.Tool -o javaextractor javaextractor/java15.g
java antlr.Tool -o javaextractor javaextractor/java15.tree.g
```

4. Dann müssen JGraLab und das darin enthaltene Programm `TGSchema2Java` zur Generierung von Graphklassen aus einem Schema ebenfalls kompiliert werden:

   ```
   javac ../../jgralab/src/de/uni_koblenz/jgralab/*.java
   javac ../../jgralab/src/de/uni_koblenz/jgralab/utilities/
       tgschema2java/*.java
   ```

5. Nun können die Graph-, Knoten- und Kantenklassen, die der Javaextraktor benötigt, erzeugt werden. `TGSchema2Java` generiert die entsprechenden Quelltexte aus den Informationen der Schemadatei (`java5.tg` im Verzeichnis `localrepository/javaextractor/src`) und schreibt diese in `localrepository/javaextractor/src/javaextractor/schema`. Der Aufruf geschieht mit folgendem Kommando:

   ```
   java de.uni_koblenz.jgralab.utilities.tgschema2java.
       TgSchema2Java -f java5.tg -p .
   ```

6. Nachdem nun alle benötigten Komponenten erzeugt und Quelltexte kompiliert wurden, wird das eigentliche Extraktorprogramm kompiliert:

   ```
   javac javaextractor/*.java
   ```

Abbildung 7: Verzeichnisstruktur (vereinfacht) im verwendeten lokalen Repository.

Nach erfolgreicher Erzeugung kann der Javaextraktor verwendet werden. Dies wird im nächsten Kapitel beschrieben.

8.2 Aufruf

Um den Javaextraktor zu verwenden, müssen die gleichen Verzeichnisse und Dateien im CLASSPATH enthalten sein, wie im vorherigen Kapitel beschrieben. Der Aufruf des Javaextraktors geschieht durch:

```
java javaextractor.JavaExtractor
```

Zusätzlich interpretiert das Programm folgende (durch Leerzeichen getrennte) Kommandozeilenparameter:

- PFAD : Übergibt den Pfad einer zu parsenden Datei oder eines Verzeichnisses. Dieser Parameter muss mindestens einmal und kann beliebig oft vorhanden sein.
- -out DATEINAME oder -o DATEINAME : Übergibt den Pfad zur Datei, in die der erzeugte TGraph gespeichert wird. Die Datei sollte die Endung .tg besitzen. Dieser Parameter ist optional, standardmäßig wird die Datei extractedgraph.tg im aktuellen Verzeichnis verwendet.
- -name PROGRAMMNAME oder -n PROGRAMMNAME : Übergibt den Namen, welcher im erzeugten TGraph als Programmname gesetzt wird. Dieser Parameter ist optional.
- -log DATEINAME oder -l DATEINAME : Übergibt den Pfad zur Datei, in die das Log geschrieben wird. Dieser Parameter ist optional, standardmäßig wird die Datei javaextractor.log im aktuellen Verzeichnis verwendet.
- -eager oder -e : Führt den Extraktor im EAGER-Modus aus. Dieser Parameter ist optional, Standard ist der LAZY-Modus.
- -complete oder -c : Führt den Extraktor im COMPLETE-Modus aus. Dieser Parameter ist optional, Standard ist der LAZY-Modus.

Zu beachten ist außerdem, dass beim Parsen einer größeren Menge Quelltext der Heap überlaufen kann. Dies lässt sich verhindern, indem Java über den Kommandozeilenparameter -Xmx ein höherer Maximalwert für den Heap zugewiesen wird.

Ein konkreter Aufruf des Extraktors mit einer maximalen Heapgröße von 768MB und unter Verwendung aller o.g. Kommandozeilenparameter sieht wie folgt aus:

```
java -Xmx768M javaextractor.JavaExtractor -out testgraph.tg -name
    Testprogramm -log testextract.log -eager ../testit/test1.java
    ../testit/test2.java ../testit2
```

9 Abschließende Betrachtung

Dieses Kapitel widmet sich einer abschließenden Betrachtung der Entwicklung des Javaextraktors. Zunächst werden die umgesetzten Anforderungen bewertet und abschließend mögliche Weiterentwicklungen aufgezeigt.

9.1 Umgesetzte Anforderungen

Die meisten Anforderungen konnten umgesetzt werden. Von den MUSS-Anforderungen konnten alle umgesetzt werden.

Nur eine Anforderung (Nr. 6) der SOLLTE-Anforderungen konnte nicht realisiert werden, da die Javagrammatik von Michael Studman nur syntaktisch korrekten Code zuließ. Die Erfüllung der Anfoderung wäre zu aufwendig gewesen, da eine neue Grammatik hätte entwickelt werden müssen.

Zwei optionale Anforderungen (Nr. 4 und 12) wurden nicht umgesetzt. Eine Beschäftigung mit der Anfragesprache für TGraphen GReQL [12] und das Vornehmen von Unit-Tests konnten nicht mehr im zeitlichen Rahmen der Studienarbeit stattfinden.

9.2 Ausblick

Im Rahmen dieser Studienarbeit wurde ein Faktenextraktor für Java-Quelltexte entwickelt. Der Javaextraktor generiert eine TGraph-Repräsentation der Quelltexte eines in Java implementierten Softwareprojekts. Dieser TGraph ist dann durch die Werkzeuge des Gupro-Projekts weiter verarbeitbar.

Der Javaextraktor kann Javaquelltexte bis einschließlich Javaversion 6 verarbeiten. Dazu wurde ein Werkzeug gefunden, welches den Parsingvorgang übernimmt, so dass diese Funktion nicht selbst implementiert werden musste. Zusätzlich wurde ein Javametamodell und entsprechendes Schema für den TGraph entwickelt. Die Knoten- und Kantentypen des Metamodells konnten automatisiert mit JGraLab aus dem Schema erzeugt werden.

In der aktuellen Version bringt der Javextraktor Metamodell und Schema zur feingranularen Abbildung der Syntax mit. Wünschenswert sind weitere Metamodelle und Schemata, die weiter von der Syntax abstrahieren. Denkbar sind Modell und Schemata für TGraphen, die z. B. ausschließlich Vererbungshierarchien oder Benutzungsbeziehungen von Klassen darstellen. Hier könnten Projekte zur Weiterentwicklungen ansetzen, da jeweils ein entsprechender Treewalker implementiert werden müsste.

Insgesamt kann die Entwicklung des Javaextraktors als positiv betrachtet werden, da dieser nicht nur die zwingend geforderten Mindesteigenschaften besitzt, sondern darüberhinaus auch eine Reihe weiterer optionaler Anforderungen erfüllt.

A Metamodell der Graphklassen

Nach dem eigentlichen Metamodell mit den Knoten- / Kantenbeziehungen und der Knotentypenhierarchie folgt die Kantentypenhierarchie in einem eigenständigen Diagramm.

Abbildung 8: Metamodell (Teil 1 / 12)

Abbildung 9: Metamodell (Teil 2 / 12)

Abbildung 10: Metamodell (Teil 3 / 12)

Abbildung 11: Metamodell (Teil 4 / 12)

Abbildung 12: Metamodell (Teil 5 / 12)

Abbildung 13: Metamodell (Teil 6 / 12)

Abbildung 14: Metamodell (Teil 7 / 12)

Abbildung 15: Metamodell (Teil 8 / 12)

Abbildung 16: Metamodell (Teil 9 / 12)

Abbildung 17: Metamodell (Teil 10 / 12)

Abbildung 18: Metamodell (Teil 11 / 12)

Expression

StringConstant
value : String

CharConstant
value : Integer
literal : String

LongConstant
value : Long
literal : String

IntegerConstant
value : Long
literal : String

DoubleConstant
value : Double
literal : String

FloatConstant
value : Double
literal : String

BooleanConstant
value : Boolean

Null

Abbildung 19: Metamodell (Teil 12 / 12)

Edge

IsDeclarationOfInvokedMethod

IsDeclarationOfAccessedField

IsTypeDefinitionOf

IsSubPackageOf

IsPartOf

AttributedEdge
offset : Integer
length : Integer
line : Integer
column : Integer

IsSourceUsageIn

IsSourceFor

IsPrimarySourceFor

IsTranslationUnitIn

IsPackageOf

IsProgramNameOf

Abbildung 20: Kantentypenhierarchie (Teil 1 / 13)

AttributedEdge
offset : Integer
length : Integer
line : Integer
column : Integer

IsLoopBodyOf
IsRightHandSideOf
IsLeftHandSideOf
IsInterfaceOf
IsSuperClassOf
IsExceptionThrownBy
IsConstructorBodyOf
IsConditionOf
IsArgumentOf
IsAnnotationOf

Abbildung 21: Kantentypenhierarchie (Teil 2 / 13)

AttributedEdge
offset : Integer
length : Integer
line : Integer
column : Integer

IsTypeParameterOf
IsTypeOf
IsBlockOf
IsStatementOf
IsParameterOf
IsModifierOf

Abbildung 22: Kantentypenhierarchie (Teil 3 / 13)

IsAnnotationOf

IsAnnotationOfVariable

IsAnnotationOfType

IsAnnotationOfEnumConstant

IsAnnotationOfPackage

IsMetaAnnotationOf

IsAnnotationOfMember

Abbildung 23: Kantentypenhierarchie (Teil 4 / 13)

IsSwitchArgumentOf
IsWildCardArgumentOf
IsSimpleArgumentOf
IsArgumentOfMethodInvocation
IsArgumentOfEnumConstant
IsArgumentOf

Abbildung 24: Kantentypenhierarchie (Teil 5 / 13)

IsConditionOf

IsCaseConditionOf

IsForConditionOf

IsConditionOfWhile

IsConditionOfDoWhile

IsConditionOfExpression

IsConditionOfIf

IsConditionOfAssert

Abbildung 25: Kantentypenhierarchie (Teil 6 / 13)

Abbildung 26: Kantentypenhierarchie (Teil 7 / 13)

IsLoopBodyOf
IsLoopBodyOfWhile
IsLoopBodyOfDoWhile
IsLoopBodyOfFor
IsRightHandSideOf
IsRightHandSideOfPrefixExpression
IsRightHandSideOfInfixExpression
IsLeftHandSideOf
IsLeftHandSideOfPostfixExpression
IsLeftHandSideOfInfixExpression

Abbildung 27: Kantentypenhierarchie (Teil 8 / 13)

Abbildung 28: Kantentypenhierarchie (Teil 9 / 13)

Abbildung 29: Kantentypenhierarchie (Teil 10 / 13)

Abbildung 30: Kantentypenhierarchie (Teil 11 / 13)

Abbildung 31: Kantentypenhierarchie (Teil 12 / 13)

Abbildung 32: Kantentypenhierarchie (Teil 13 / 13)

B Schema der Graphklassen

```
1  Schema javaextractor.schema.Java5Schema ;
2
3  EnumDomain InfixOperators( ASSIGNMENT,
4                             EQUALS,
5                             PLUS,
6                             MINUS,
7                             MULTIPLICATION,
8                             DIVISION,
9                             AND,
10                            SHORTCIRCUITAND,
11                            OR,
12                            SHORTCIRCUITOR,
13                            MODULO,
14                            LEFTSHIFT,
15                            RIGHTSHIFT,
16                            UNSIGNEDRIGHTSHIFT,
17                            UNEQUALS,
18                            XOR,
19                            PLUSASSIGNMENT,
20                            MINUSASSIGNMENT,
21                            MULTIPLICATIONASSIGNMENT,
22                            DIVISIONASSIGNMENT,
23                            GREATER,
24                            GREATEREQUALS,
25                            LESS,
26                            LESSEQUALS,
27                            MODULOASSIGNMENT,
28                            LEFTSHIFTASSIGNMENT,
29                            RIGHTSHIFTASSIGNMENT,
30                            UNSIGNEDRIGHTSHIFTASSIGNMENT,
31                            ANDASSIGNMENT,
32                            ORASSIGNMENT,
33                            XORASSIGNMENT,
34                            INSTANCEOF ) ;
35
36 EnumDomain PrefixOperators( PLUS,
37                             MINUS,
38                             NOT,
39                             BITWISECOMPLEMENT,
40                             INCREMENT,
41                             DECREMENT ) ;
42
```

```
43 EnumDomain PostfixOperators( DECREMENT,
44                              INCREMENT ) ;
45
46 EnumDomain BuiltInTypes( VOID,
47                          BOOLEAN,
48                          BYTE,
49                          CHAR,
50                          SHORT,
51                          INT,
52                          LONG,
53                          FLOAT,
54                          DOUBLE ) ;
55
56 EnumDomain Modifiers( PUBLIC,
57                       PROTECTED,
58                       PRIVATE,
59                       STATIC,
60                       FINAL,
61                       ABSTRACT,
62                       VOLATILE,
63                       SYNCHRONIZED,
64                       NATIVE,
65                       STRICTFP,
66                       TRANSIENT ) ;
67
68 EnumDomain MethodInvocationTypes( METHOD,
69                                   CONSTRUCTOR,
70                                   SUPERMETHOD,
71                                   SUPERCONSTRUCTOR,
72                                   EXPLICITCONSTRUCTOR ) ;
73
74 GraphClass Java5 { name : String, version : String } ;
75
76 abstract EdgeClass AttributedEdge from Vertex( 0, ∗ ) to Vertex( 0, ∗ ) { offset : Integer,
       length : Integer , line : Integer , column : Integer } ;
77
78 abstract VertexClass Type : ExternalDeclaration, Member { name : String,
       fullyQualifiedName : String, external : Boolean } ;
79 abstract VertexClass Member ;
80 abstract VertexClass TypeSpecification ;
81 abstract VertexClass Statement ;
82 abstract VertexClass Expression : Statement ;
83 abstract VertexClass ForHead ;
84 abstract VertexClass FieldDeclaration ;
```

```
85
86 VertexClass Annotation ;
87 VertexClass Block : Statement;
88 VertexClass Identifier { name : String } ;
89 VertexClass Modifier { type : Modifiers } ;
90 VertexClass TypeParameterDeclaration : Type ;
91 VertexClass ParameterDeclaration : FieldDeclaration ;
92 VertexClass TypeArgument : TypeSpecification ;
93 VertexClass MethodDeclaration : Member ;
94
95 abstract EdgeClass IsAnnotationOf : AttributedEdge from Annotation( 0, * ) to Vertex( 1,
     1 ) ;
96 abstract EdgeClass IsArgumentOf : AttributedEdge from Vertex( 0, * ) to Vertex( 1, 1 ) ;
97 abstract EdgeClass IsConditionOf : AttributedEdge from Expression( 0, 1 ) to Vertex( 1,
     1 ) ;
98 abstract EdgeClass IsExceptionThrownBy : AttributedEdge from TypeSpecification( 0, *
     ) to Member( 1, 1 ) ;
99 abstract EdgeClass IsInterfaceOf : AttributedEdge from TypeSpecification( 0, * ) to Type
     ( 1, 1 ) ;
100 abstract EdgeClass IsSuperClassOf : AttributedEdge from TypeSpecification( 0, 1 ) to
     Type( 1, 1 ) ;
101 abstract EdgeClass IsLeftHandSideOf : AttributedEdge from Expression( 1, 1 ) to
     Expression( 1, 1 ) ;
102 abstract EdgeClass IsRightHandSideOf : AttributedEdge from Expression( 1, 1 ) to
     Expression( 1, 1 ) ;
103 abstract EdgeClass IsLoopBodyOf : AttributedEdge from Statement( 1, 1 ) to Statement
     ( 1, 1 ) ;
104 abstract EdgeClass IsModifierOf : AttributedEdge from Modifier( 0, * ) to Vertex( 1, 1 ) ;
105 abstract EdgeClass IsParameterOf : AttributedEdge from ParameterDeclaration( 0, * )
     to Vertex( 1, 1 ) ;
106 abstract EdgeClass IsStatementOf : AttributedEdge from Statement( 0, * ) to Vertex( 1,
     1 ) ;
107 abstract EdgeClass IsTypeOf : AttributedEdge from Vertex( 0, * ) to Vertex( 1, 1 ) ;
108 abstract EdgeClass IsTypeParameterOf : AttributedEdge from Vertex( 0, * ) to Vertex( 1,
     1 ) ;
109 abstract EdgeClass IsBlockOf : AttributedEdge from Block( 1, 1 ) to Vertex( 1, 1  );
110
111 EdgeClass IsTypeArgumentOfTypeSpecification : AttributedEdge from TypeArgument( 0,
     * ) to TypeSpecification( 1, 1 ) ;
112
113 VertexClass Program { name : String } ;
114 VertexClass TranslationUnit ;
115 EdgeClass IsTranslationUnitIn from TranslationUnit( 0, * ) to Program( 1, 1 ) ;
116 abstract VertexClass FoldGraphReference { lengthOfFile : Integer } ;
```

```
117 VertexClass SourceUsage : FoldGraphReference ;
118 EdgeClass IsSourceUsageIn from SourceUsage( 1, 1 ) to TranslationUnit( 1, 1 );
119 VertexClass SourceFile { name: String } ;
120 EdgeClass IsPrimarySourceFor from SourceFile( 1, 1 ) to TranslationUnit( 1, 1 ) ;
121 EdgeClass IsSourceFor from SourceFile( 1, 1 ) to SourceUsage( 1, 1 ) ;
122
123 abstract VertexClass ExternalDeclaration ;
124 EdgeClass IsExternalDeclarationIn : AttributedEdge from ExternalDeclaration( 0, * ) to
        SourceUsage( 1, 1 ) ;
125
126 abstract VertexClass ImportDefinition : ExternalDeclaration ;
127 VertexClass ClassImportDefinition : ImportDefinition ;
128 VertexClass PackageImportDefinition : ImportDefinition ;
129
130 VertexClass JavaPackage { fullyQualifiedName : String } ;
131 VertexClass PackageDefinition : ExternalDeclaration;
132 EdgeClass IsPackageNameOf : AttributedEdge from QualifiedName( 1, 1 ) to
        PackageDefinition( 1,* );
133 EdgeClass IsPartOf from TranslationUnit( 0, * ) to JavaPackage( 1, 1 ) ;
134 EdgeClass IsPackageOf from JavaPackage( 1, * ) to Program( 1, 1 ) ;
135 EdgeClass IsSubPackageOf from JavaPackage( 0, 1 ) to JavaPackage( 1, 1 ) ;
136
137 VertexClass Comment ;
138 VertexClass SingleLineComment : Comment ;
139 VertexClass MultiLineComment : Comment ;
140 VertexClass JavaDocComment : Comment ;
141 EdgeClass IsCommentIn : AttributedEdge from Comment( 0, * ) to TranslationUnit( 1, 1 )
        ;
142
143 EdgeClass IsAnnotationNameOf : AttributedEdge from QualifiedName( 1, 1 ) to
        Annotation( 1, 1 ) ;
144 EdgeClass IsAnnotationArgumentOf : AttributedEdge from Expression( 0, * ) to
        Annotation( 1, 1 ) ;
145
146 EdgeClass IsAnnotationOfPackage : IsAnnotationOf from Annotation( 0, * ) to
        PackageDefinition( 1, 1 ) ;
147
148 EdgeClass IsStatementOfBody : IsStatementOf from Statement( 0, * ) to Block( 1, 1 ) ;
149 EdgeClass IsTypeIn : AttributedEdge from Type( 0, * ) to Statement( 1, 1 ) ;
150
151 VertexClass SimpleArgument ;
152 VertexClass WildcardArgument ;
153 EdgeClass IsSimpleArgumentOf : IsArgumentOf from SimpleArgument( 0, * ) to
        TypeArgument( 1, 1 ) ;
```

```
154 EdgeClass IsWildcardArgumentOf : IsArgumentOf from WildcardArgument( 0, ∗ ) to
        TypeArgument( 1, 1 ) ;
155
156 EdgeClass IsTypeOfSimpleArgument : IsTypeOf from TypeSpecification( 1, 1 ) to
        SimpleArgument( 1, 1 ) ;
157
158 EdgeClass IsLowerBoundOfWildcardArgument : AttributedEdge from TypeSpecification(
        0, 1 ) to WildcardArgument( 1, 1 ) ;
159 EdgeClass IsUpperBoundOfWildcardArgument : AttributedEdge from TypeSpecification(
        0, 1 ) to WildcardArgument( 1, 1 ) ;
160
161 VertexClass TypeParameterUsage : TypeSpecification ;
162 EdgeClass IsTypeParameterUsageNameOf : AttributedEdge from Identifier( 1, 1 ) to
        TypeParameterUsage( 1, 1 ) ;
163
164 VertexClass ArrayType : TypeSpecification { dimensions : Integer } ;
165 EdgeClass IsElementTypeOf : IsTypeOf from TypeSpecification( 1, 1 ) to ArrayType( 1, 1
        ) ;
166
167 VertexClass QualifiedName : TypeSpecification { fullyQualifiedName : String } ;
168 VertexClass QualifiedType : QualifiedName, Expression ;
169 EdgeClass IsQualifiedNameOf : AttributedEdge from QualifiedName( 0, ∗ ) to Vertex( 0,
        ∗ ) ;
170 EdgeClass IsNameOf : AttributedEdge from Identifier( 0, ∗ ) to QualifiedName( 0, ∗ ) ;
171 EdgeClass IsImportedTypeOf : AttributedEdge from QualifiedName( 1, 1 ) to
        ImportDefinition( 1, 1 ) ;
172 EdgeClass IsTypeDefinitionOf from Type( 0, ∗ ) to TypeSpecification( 1, 1 );
173
174 VertexClass BuiltInType : TypeSpecification { type : BuiltInTypes } ;
175
176 EdgeClass IsTypeParameterDeclarationNameOf : AttributedEdge from Identifier( 1, 1 ) to
        TypeParameterDeclaration( 1, 1 ) ;
177 EdgeClass IsUpperBoundOfTypeParameter : AttributedEdge from TypeSpecification( 0, ∗
        ) to TypeParameterDeclaration( 1, 1 ) ;
178
179 EdgeClass IsAnnotationOfType : IsAnnotationOf from Annotation( 0, ∗ ) to Type( 1, 1 ) ;
180
181 VertexClass ClassDefinition : Type ;
182 EdgeClass IsModifierOfClass : IsModifierOf from Modifier( 0, ∗ ) to ClassDefinition( 1, 1 )
        ;
183 EdgeClass IsClassNameOf : AttributedEdge from Identifier( 1, 1 ) to ClassDefinition( 1, 1
        ) ;
184 EdgeClass IsTypeParameterOfClass : IsTypeParameterOf from
        TypeParameterDeclaration( 0, ∗ ) to ClassDefinition( 1, 1 ) ;
```

```
185 EdgeClass IsSuperClassOfClass : IsSuperClassOf from TypeSpecification( 0, 1 ) to
      ClassDefinition( 1, 1 ) ;
186 EdgeClass IsInterfaceOfClass : IsInterfaceOf from TypeSpecification( 0, ∗ ) to
      ClassDefinition( 1, 1 ) ;
187 EdgeClass IsClassBlockOf : IsBlockOf from Block( 1, 1 ) to ClassDefinition( 1, 1 ) ;
188
189 VertexClass InterfaceDefinition : Type ;
190 EdgeClass IsModifierOfInterface : IsModifierOf from Modifier( 0, ∗ ) to InterfaceDefinition(
      1, 1 ) ;
191 EdgeClass IsInterfaceNameOf : AttributedEdge from Identifier( 1, 1 ) to
      InterfaceDefinition( 1, 1 ) ;
192 EdgeClass IsTypeParameterOfInterface : IsTypeParameterOf from
      TypeParameterDeclaration( 0, ∗ ) to InterfaceDefinition( 1, 1 ) ;
193 EdgeClass IsSuperClassOfInterface : IsSuperClassOf from TypeSpecification( 0, ∗ ) to
      InterfaceDefinition( 1, 1 ) ;
194 EdgeClass IsInterfaceBlockOf : IsBlockOf from Block( 1, 1 ) to InterfaceDefinition( 1, 1 ) ;
195
196
197 VertexClass VariableDeclaration : FieldDeclaration, Statement ;
198
199 VertexClass EnumConstant : Member, FieldDeclaration ;
200 EdgeClass IsAnnotationOfEnumConstant : IsAnnotationOf from Annotation( 1, 1 ) to
      EnumConstant( 1, 1 ) ;
201 EdgeClass IsEnumConstantNameOf : AttributedEdge from Identifier( 1, 1 ) to
      EnumConstant( 1, 1 ) ;
202 EdgeClass IsArgumentOfEnumConstant : IsArgumentOf from Expression( 0, ∗ ) to
      EnumConstant( 1, 1 ) ;
203 EdgeClass IsEnumConstantBlockOf : IsBlockOf from Block( 1, 1 ) to EnumConstant( 1, 1
      );
204
205 VertexClass EnumDefinition : Type ;
206 EdgeClass IsModifierOfEnum : IsModifierOf from Modifier( 0, ∗ ) to EnumDefinition( 1, 1
      ) ;
207 EdgeClass IsEnumNameOf : AttributedEdge from Identifier( 1, 1 ) to EnumDefinition( 1,
      1 ) ;
208 EdgeClass IsInterfaceOfEnum : IsInterfaceOf from TypeSpecification( 0, ∗ ) to
      EnumDefinition( 1, 1 ) ;
209 EdgeClass IsEnumBlockOf : IsBlockOf from Block( 1, 1 ) to EnumDefinition( 1, 1 ) ;
210
211 VertexClass AnnotationDefinition : Type ;
212 EdgeClass IsModifierOfAnnotation : IsModifierOf from Modifier( 0, ∗ ) to
      AnnotationDefinition( 1, 1 ) ;
213 EdgeClass IsAnnotationDefinitionNameOf : AttributedEdge from Identifier( 1, 1 ) to
      AnnotationDefinition( 1, 1 ) ;
```

```
214 EdgeClass IsMetaAnnotationOf : IsAnnotationOf from Annotation( 0, * ) to
      AnnotationDefinition( 1, 1 ) ;
215 EdgeClass IsAnnotationBlockOf : IsBlockOf from Block( 1, 1 ) to AnnotationDefinition( 1,
      1 ) ;
216
217 VertexClass FieldAccess : Expression ;
218 EdgeClass IsFieldContainerOf : AttributedEdge from Expression( 0, 1 ) to FieldAccess(
      1, 1 ) ;
219 EdgeClass IsFieldNameOf : AttributedEdge from Identifier( 1, 1 ) to FieldAccess( 1, 1 ) ;
220 EdgeClass IsArrayElementIndexOf : AttributedEdge from Expression( 0, * ) to
      FieldAccess( 1, 1 ) ;
221 EdgeClass IsDeclarationOfAccessedField from Vertex( 0, * ) to FieldAccess( 1, 1 ) ;
222
223 VertexClass MethodInvocation : Expression { type : MethodInvocationTypes } ;
224 EdgeClass IsMethodContainerOf : AttributedEdge from Expression( 0, 1 ) to
      MethodInvocation( 1, 1 ) ;
225 EdgeClass IsNameOfInvokedMethod : AttributedEdge from Identifier( 1, 1 ) to
      MethodInvocation( 1, 1 ) ;
226 EdgeClass IsArgumentOfMethodInvocation : IsArgumentOf from Expression( 0, * ) to
      MethodInvocation( 1, 1 ) ;
227 EdgeClass IsDeclarationOfInvokedMethod from Member( 0, * ) to MethodInvocation( 1,
      1 ) ;
228
229 VertexClass ConditionalExpression : Expression ;
230 EdgeClass IsConditionOfExpression : IsConditionOf from Expression( 1, 1 ) to
      ConditionalExpression( 1, 1 ) ;
231 EdgeClass IsMatchOf : AttributedEdge from Expression( 1, 1 ) to ConditionalExpression(
       1, 1 ) ;
232 EdgeClass IsMismatchOf : AttributedEdge from Expression( 1, 1 ) to
      ConditionalExpression( 1, 1 ) ;
233
234 VertexClass BuiltInCast : Expression ;
235 EdgeClass IsCastedBuiltInTypeOf : IsTypeOf from BuiltInType( 1, 1 ) to BuiltInCast( 1, 1 )
       ;
236 EdgeClass IsCastedValueOf : AttributedEdge from Expression( 1, 1 ) to BuiltInCast( 1, 1
      ) ;
237
238 VertexClass ClassCast : Expression ;
239 EdgeClass IsCastedTypeOf : IsTypeOf from TypeSpecification( 1, 1 ) to ClassCast( 1, 1 )
      ;
240 EdgeClass IsCastedObjectOf : AttributedEdge from Expression( 1, 1 ) to ClassCast( 1, 1
       );
241
242 VertexClass InfixExpression : Expression { operator :  InfixOperators } ;
```

```
243 EdgeClass IsLeftHandSideOfInfixExpression : IsLeftHandSideOf from Expression( 1, 1 )
        to InfixExpression( 1, 1 ) ;
244 EdgeClass IsRightHandSideOfInfixExpression : IsRightHandSideOf from Expression( 1,
        1 ) to InfixExpression( 1, 1 ) ;
245
246 VertexClass PrefixExpression : Expression { operator :  PrefixOperators } ;
247 EdgeClass IsRightHandSideOfPrefixExpression : IsRightHandSideOf from Expression(
        1, 1 ) to PrefixExpression( 1, 1 ) ;
248
249 VertexClass PostfixExpression: Expression { operator :  PostfixOperators } ;
250 EdgeClass IsLeftHandSideOfPostfixExpression : IsLeftHandSideOf from Expression( 1,
        1 ) to PostfixExpression( 1, 1 ) ;
251
252 VertexClass Null : Expression ;
253
254 VertexClass BooleanConstant : Expression { value : Boolean } ;
255
256 VertexClass FloatConstant : Expression { value : Double, literal : String } ;
257
258 VertexClass DoubleConstant : Expression { value : Double, literal : String } ;
259
260 VertexClass IntegerConstant : Expression { value : Long, literal : String } ;
261
262 VertexClass LongConstant : Expression { value : Long, literal : String } ;
263
264 VertexClass CharConstant : Expression { value : Integer, literal : String } ;
265
266 VertexClass StringConstant : Expression { value : String } ;
267
268 VertexClass ArrayInitializer : Expression ;
269 EdgeClass IsSizeOf : AttributedEdge from Expression( 0, 1 ) to ArrayInitializer( 1, 1 );
270 EdgeClass IsContentOf : AttributedEdge from Expression( 0, * ) to ArrayInitializer( 1, 1 );
271
272 VertexClass ArrayCreation : Expression ;
273 EdgeClass IsElementTypeOfCreatedArray : IsTypeOf from TypeSpecification( 0, 1 ) to
        ArrayCreation( 1, 1 ) ;
274 EdgeClass IsDimensionInitializerOf : AttributedEdge from ArrayInitializer( 1, * ) to
        ArrayCreation( 1, 1 ) ;
275
276 VertexClass ObjectCreation : Expression ;
277 EdgeClass IsTypeOfObject : AttributedEdge from TypeSpecification( 1, 1 ) to
        ObjectCreation( 1, 1 );
278 EdgeClass IsConstructorInvocationOf : AttributedEdge from MethodInvocation( 1, 1 ) to
        ObjectCreation( 1, 1 ) ;
```

```
279
280 VertexClass VariableInitializer : Expression ;
281 EdgeClass IsInitializerOf : AttributedEdge from Expression( 1, 1 ) to VariableInitializer( 1,
      1 ) ;
282
283 EdgeClass IsAnnotationOfMember : IsAnnotationOf from Annotation( 0, * ) to Member(
      1, 1 ) ;
284 EdgeClass IsMemberOf : AttributedEdge from Member( 0, * ) to Block( 1, 1 ) ;
285
286 VertexClass Field : Member ;
287 EdgeClass IsFieldCreationOf : AttributedEdge from Statement( 1, 1 ) to Field( 1, 1 ) ;
288
289 VertexClass AnnotationField : Member ;
290 EdgeClass IsModifierOfAnnotationField : IsModifierOf from Modifier( 0, * ) to
      AnnotationField( 1, 1 ) ;
291 EdgeClass IsTypeOfAnnotationField : AttributedEdge from TypeSpecification( 1, 1 ) to
      AnnotationField( 1, 1 );
292 EdgeClass IsAnnotationFieldNameOf : AttributedEdge from Identifier( 1, 1 ) to
      AnnotationField( 1, 1 );
293 EdgeClass IsDefaultValueOf : AttributedEdge from Expression( 1, 1 ) to AnnotationField(
      1, 1 );
294
295 VertexClass ConstructorDefinition : Member ;
296 EdgeClass IsModifierOfConstructor : IsModifierOf from Modifier( 0, * ) to
      ConstructorDefinition( 1, 1 ) ;
297 EdgeClass IsTypeParameterOfConstructor : IsTypeParameterOf from
      TypeParameterDeclaration( 0, * ) to ConstructorDefinition( 1, 1 ) ;
298 EdgeClass IsNameOfConstructor : AttributedEdge from Identifier( 1, 1 ) to
      ConstructorDefinition( 1, 1 ) ;
299 EdgeClass IsParameterOfConstructor : IsParameterOf from ParameterDeclaration( 0, * )
      to ConstructorDefinition( 1, 1 ) ;
300 EdgeClass IsExceptionThrownByConstructor : IsExceptionThrownBy from
      TypeSpecification( 0, * ) to ConstructorDefinition( 1, 1 ) ;
301 EdgeClass IsBodyOfConstructor : AttributedEdge from Block( 1, 1 ) to
      ConstructorDefinition( 1, 1 ) ;
302
303 VertexClass StaticInitializerDefinition : Member ;
304 EdgeClass IsBodyOfStaticInitializer : AttributedEdge from Block( 1, 1 ) to
      StaticInitializerDefinition( 1, 1 ) ;
305
306 VertexClass StaticConstructorDefinition : Member ;
307 EdgeClass IsBodyOfStaticConstructor : AttributedEdge from Block( 1, 1 ) to
      StaticConstructorDefinition( 1, 1 ) ;
308
```

```
309 EdgeClass IsModifierOfMethod : IsModifierOf from Modifier( 0, ∗ ) to MethodDeclaration(
      1, 1 ) ;
310 EdgeClass IsReturnTypeOf : IsTypeOf from TypeSpecification( 1, 1 ) to
      MethodDeclaration( 1, 1 ) ;
311 EdgeClass IsTypeParameterOfMethod : IsTypeParameterOf from
      TypeParameterDeclaration( 0, ∗ ) to MethodDeclaration( 1, 1 ) ;
312 EdgeClass IsNameOfMethod : AttributedEdge from Identifier( 1, 1 ) to
      MethodDeclaration( 1, 1 ) ;
313 EdgeClass IsParameterOfMethod : IsParameterOf from ParameterDeclaration( 0, ∗ ) to
      MethodDeclaration( 1, 1 ) ;
314 EdgeClass IsExceptionThrownByMethod : IsExceptionThrownBy from TypeSpecification(
      0, ∗ ) to MethodDeclaration( 1, 1 ) ;
315
316 VertexClass MethodDefinition : MethodDeclaration ;
317 EdgeClass IsBodyOfMethod : AttributedEdge from Block( 1, 1 ) to MethodDefinition( 1, 1
      ) ;
318
319 EdgeClass IsModifierOfParameter : IsModifierOf from Modifier( 0, 1 ) to
      ParameterDeclaration( 1, 1 ) ;
320 EdgeClass IsTypeOfParameter : IsTypeOf from TypeSpecification( 1, 1 ) to
      ParameterDeclaration( 1, 1 ) ;
321 EdgeClass IsParameterNameOf : AttributedEdge from Identifier( 1, 1 ) to
      ParameterDeclaration( 1, 1 ) ;
322 VertexClass VariableLengthDeclaration : ParameterDeclaration ;
323
324 VertexClass Label : Statement ;
325 EdgeClass IsLabelNameOf : AttributedEdge from Identifier( 1, 1 ) to Label( 1, 1 ) ;
326 EdgeClass IsAttachedTo : IsStatementOf from Statement( 1, 1 ) to Label( 1, 1 ) ;
327
328 VertexClass If : Statement ;
329 EdgeClass IsConditionOfIf : IsConditionOf from Expression( 1, 1 ) to If( 1, 1 ) ;
330 EdgeClass IsThenOf : IsStatementOf from Statement( 1, 1 ) to If( 1, 1 ) ;
331 EdgeClass IsElseOf : IsStatementOf from Statement( 0, 1 ) to If( 1, 1 ) ;
332
333 VertexClass For : Statement ;
334 EdgeClass IsHeadOfFor : AttributedEdge from ForHead( 1, 1 ) to For( 1, 1 ) ;
335 EdgeClass IsLoopBodyOfFor : IsLoopBodyOf from Statement( 1, 1 ) to For( 1, 1 ) ;
336 VertexClass TraditionalForClause : ForHead ;
337 EdgeClass IsRunVariableInitializationOf : AttributedEdge from Statement( 0, ∗ ) to
      TraditionalForClause( 1, 1 ) ;
338 EdgeClass IsForConditionOf : IsConditionOf from Expression( 0, 1 ) to
      TraditionalForClause( 1, 1 ) ;
339 EdgeClass IsIteratorOf : AttributedEdge from Expression( 0, ∗ ) to TraditionalForClause(
      1, 1 ) ;
```

```
340 VertexClass ForEachClause : ForHead;
341 EdgeClass IsParameterOfForEachClause : IsParameterOf from ParameterDeclaration( 1,
      1 ) to ForEachClause( 1, 1 ) ;
342 EdgeClass IsEnumerableOf : AttributedEdge from Expression( 1, 1 ) to ForEachClause(
      1, 1 ) ;
343
344 VertexClass EmptyStatement : Statement ;
345
346 VertexClass While : Statement ;
347 EdgeClass IsLoopBodyOfWhile : IsLoopBodyOf from Statement( 1, 1 ) to While( 1, 1 ) ;
348 EdgeClass IsConditionOfWhile : IsConditionOf from Expression( 1, 1 ) to While( 1, 1 ) ;
349
350 VertexClass DoWhile : Statement ;
351 EdgeClass IsLoopBodyOfDoWhile : IsLoopBodyOf from Statement( 1, 1 ) to DoWhile( 1,
      1 ) ;
352 EdgeClass IsConditionOfDoWhile : IsConditionOf from Expression( 1, 1 ) to DoWhile( 1,
      1 ) ;
353
354 VertexClass Break : Statement ;
355 EdgeClass IsBreakTargetOf : AttributedEdge from Label( 0, 1 ) to Break( 1, 1 ) ;
356
357 VertexClass Continue : Statement ;
358 EdgeClass IsContinueTargetOf : AttributedEdge from Label( 0, 1 ) to Continue( 1, 1 ) ;
359
360 VertexClass Return : Statement ;
361 EdgeClass IsReturnedBy : AttributedEdge from Expression( 0, 1 ) to Return( 1, 1 ) ;
362
363 VertexClass Switch : Statement ;
364 VertexClass Case : Statement ;
365 VertexClass Default : Statement ;
366 EdgeClass IsSwitchArgumentOf : IsArgumentOf from Expression( 1, 1 ) to Switch( 1, 1 )
      ;
367 EdgeClass IsCaseOf : AttributedEdge from Case( 0, * ) to Switch( 1, 1 ) ;
368 EdgeClass IsDefaultCaseOf : AttributedEdge from Default( 0, 1 ) to Switch( 1, 1 ) ;
369 EdgeClass IsCaseConditionOf : IsConditionOf from Expression( 1, 1 ) to Case( 1, 1 ) ;
370 EdgeClass IsStatementOfCase : IsStatementOf from Statement( 0, * ) to Case( 1, 1 ) ;
371 EdgeClass IsStatementOfDefaultCase : IsStatementOf from Statement( 0, * ) to Default(
      1, 1 ) ;
372
373 VertexClass Try : Statement ;
374 VertexClass Catch : Statement ;
375 EdgeClass IsBodyOfTry : AttributedEdge from Block( 1, 1 ) to Try( 1, 1 ) ;
376 EdgeClass IsHandlerOf : AttributedEdge from Catch( 0, * ) to Try( 1, 1 ) ;
```

```
377 EdgeClass IsCaughtExceptionOf : AttributedEdge from ParameterDeclaration( 1, 1 ) to
        Catch( 1, 1 ) ;
378 EdgeClass IsBodyOfCatch : AttributedEdge from Block( 1, 1 ) to Catch( 1, 1 ) ;
379 EdgeClass IsBodyOfFinally : AttributedEdge from Block( 0, 1 ) to Try( 1, 1 ) ;
380
381 VertexClass Throw : Statement ;
382 EdgeClass IsThrownExceptionOf : AttributedEdge from Expression( 1, 1 ) to Throw( 1, 1
        ) ;
383
384 VertexClass Synchronized : Statement ;
385 EdgeClass IsMonitorOf : AttributedEdge from Expression( 1, 1 ) to Synchronized( 1, 1 ) ;
386 EdgeClass IsSynchronizedBodyOf : AttributedEdge from Block( 1, 1 ) to Synchronized(
        1, 1 ) ;
387
388 VertexClass Assert : Statement ;
389 EdgeClass IsConditionOfAssert : IsConditionOf from Expression( 1, 1 ) to Assert( 1, 1 ) ;
390 EdgeClass IsMessageOf : AttributedEdge from Expression( 0, 1 ) to Assert( 1, 1 ) ;
391
392 EdgeClass IsAnnotationOfVariable : IsAnnotationOf from Annotation( 0, * ) to
        VariableDeclaration( 1, 1 ) ;
393 EdgeClass IsModifierOfVariable : IsModifierOf from Modifier( 0, * ) to
        VariableDeclaration( 1, 1 ) ;
394 EdgeClass IsTypeOfVariable : IsTypeOf from TypeSpecification( 1, 1 ) to
        VariableDeclaration( 1, 1 ) ;
395 EdgeClass IsVariableNameOf : AttributedEdge from Identifier( 1, * ) to
        VariableDeclaration( 1, 1 ) ;
396 EdgeClass IsInitializerOfVariable : AttributedEdge from Expression ( 0, 1 ) to
        VariableDeclaration( 1, 1 ) ;
```

Listing 1: Das im Javaextraktor eingesetzte Schema

C Dokumentation der Onlinerecherche

C.1 Vorgehen

C.1.1 Grober Ablauf der Tests

Größtenteils fanden wir für unsere Zwecke im Internet Parsergeneratoren (plus Grammatiken) und Werkzeuge, die eben diese benutzen. Im Groben haben wir die Tools wie folgt behandelt:

Zunächst haben wir versucht die Software einzurichten. Dies reichte von einer vollautomatischen Installation mittels Setup, über einen einfachen Kommandozeilenaufruf, bis zur manuellen Kompilierung der Quelltexte. Dabei haben wir festgehalten, ob dies wie vom Hersteller angegeben funktionierte, oder wir einen eigenen Weg finden mussten. Ein Produkt, welches sich überhaupt nicht einrichten ließ, haben wir nicht angetroffen.

Anschließend haben wir versucht aus einer einfachen Beispielgrammatik einen Parser zu erzeugen und diesen zu testen. Auch hier hielten wir fest, ob dies wie dokumentiert funktionierte oder ob wir einen eigenen Weg finden mussten. Teilweise stellten wir fest, dass die Erzeugung zwar noch dokumentiert war, die Verwendung der erzeugten Parser hingegen nicht mehr - und nicht immer fanden wir eine Lösung.

Im letzten Schritt haben wir versucht aus den Javagrammatiken die Parser zu erzeugen und diese zu testen. Auch hier haben wir unser Vorgehen und die Ergebnisse dokumentiert. Leider mussten wir auch hier feststellen, dass die Erzeugung noch ausreichend dokumentiert war, die Verwendung der Parser aber nicht. Fanden wir keinen eigenen Weg diese zu benutzen, entfielen die Parsingtests, auf die wir im folgenden Abschnitt genauer eingehen.

C.1.2 Parsingtests

Sinn der Parsingtests war es, zu erfahren ob die Parser:

- die neuen Java 5 Ausdrücke verarbeiten können
- auch komplizierte Javaausdrücke akzeptieren
- untypische Fehler bemerken
- mit problematischen Newlines klarkommen

Dazu haben wir die Tests in drei Phasen unterteilt.

In der ersten Phase wurden nur Quelltexte (Übersicht siehe Tabelle 4) geparst, die vom Java Compiler `javac` fehlerfrei übersetzt werden können.

Dazu mussten die Parser zunächst eine Klasse verarbeiten, die (laut Java Sprachspezifikation bis Version 1.4) grenzwertige, dennoch erlaubte Ausdrücke enthält.

Als nächstes testeten wir, ob die Parser mit Unicode-Escapesequenzen umgehen können.

Anschließend mussten die Parser Klassen verarbeiten, die jeweils nur einen Java 5 Ausdruck benutzen. Darauf aufbauend mussten diese zeigen, dass sie auch Klassen akzeptieren, die aus vielen komplizierten Kombinationen von Java 5 Ausdrücken bestehen (diese haben wir den Testquelltexten von CUP entnommen).

In der zweiten Phase wurden nur Klassen (Übersicht siehe Tabelle 5) geparst, die vom Java Compiler javac nicht fehlerfrei übersetzt werden können. Dazu haben wir Klassen genommen, die nicht aufgrund des Typecheckings, sondern aufgrund ungültiger Ausdrücken nicht kompiliert werden können. Dabei handelt es sich u. a. um ungültige Kombinationen von Modifiern und um das `for each ( .. in .. )`-Konstrukt, welches in einer Preview von Java 5 noch erlaubt war, aus Kompatibilitätsgründen aber nicht mehr in der endgültigen Fassung enthalten ist.

In der dritten Phase schließlich sollten die Parser zeigen, dass sie mit den unterschiedlichen Zeilenumbrüchen der Betriebssystemwelt umgehen können. Dazu haben wir, für jede Art von Zeilenumbruch, den Parsern eine separate Klasse (Übersicht siehe Tabelle 6), sowie eine mit der Kombination aller Umbrüche, zum Parsen gegeben.

Testquelltexte	**Inhalt**	**Listing**
Eric	Grenzwertige, dennoch erlaubte Ausdrücke bis Java 1.4	4
EscapeSequences	Unicode Escapesequenzen	5
AllowedGenericConstruct	Erlaubter generic-Ausdruck aus Java 5	6
AllowedForEachConstructs	Erlaubter ForEach-Ausdruck aus Java 5	7
AnotherAllowedForEachConstruct	Erlaubter ForEach-Ausdruck aus Java 5	8
Test15	Komplizierte Kombinationen von Java 5 Ausdrücken	9
TestJSR201Berichtigt	Komplizierte Kombinationen von Java 5 Ausdrücken	10

Tabelle 4: Kurze Beschreibung der Quelltexte für erste Parsingphase

Testquelltexte	Inhalt	Listing
ForEachInShouldGiveParseError	Nicht mehr erlaubter ForEachIn-Ausdruck	13
AbstractEnumShouldGiveParseError	Nicht erlaubter abstarkter Enum-Ausdruck	11
AbstractFinalShouldGiveParseError	Ungültige Kombination von abstract und final	12
PublicPrivateShouldGiveParseError	Ungültige Kombination von public und private	14

Tabelle 5: Kurze Beschreibung der Quelltexte für zweite Parsingphase

Testquelltexte	Inhalt
TestsPositionsLinux	Newlines Linux
TestsPositionsMac	Newlines Mac
TestsPositionsWindows	Newlines Windows
TestZeilenumbrueche-ASCII	Kombination der Newlines, Datei ASCII-kodiert

Tabelle 6: Kurze Beschreibung der Quelltexte für dritte Parsingphase

C.2 Parsergeneratoren

Bei einem Parsergenerator handelt es sich um ein Programm, welches aus einer Grammatik einen Parser erzeugt, der die Sprache, die in der Grammatik beschrieben wird, parsen kann. Da wir einen Parser für Java benötigen, werden in diesem Abschnitt die in Frage kommenden Parsergeneratoren genauer untersucht.

C.2.1 JavaCC

Eine kurze Übersicht zu den wichtigsten Daten von JavaCC findet sich in Tabelle 8.

Historie. Der Java Compiler Compiler wurde 1996 von Sreeni Viswanadha und Sriram Sankar, für die Sun Microsystems Laboratories, entwickelt. Seitdem ist das Werkzeug von den beiden stetig weiterentwickelt worden.

Seit Juni 2003 steht der Quelltext unter der FreeBSD Lizenz frei zur Verfügung und die aktuelle JavaCC-Version wurde im Januar 2006 veröffentlicht.

Charakterisierung. JavaCC ist ein in Java implementierter Parsergenerator, der LL(k)-Parser und die zugehörigen Lexer in Java erzeugt. Eine Vielzahl von Grammatiken für verschiedene Programmiersprachen ist auf [22] und [23] verfügbar (auch für Java bis Version 5). Die Erzeugung von Bäumen geschieht nicht automatisch, sondern muss von Hand in die Grammatiken eingebaut werden. Alternativ kann dies auch durch den Einsatz von Werkzeugen, wie dem mitgelieferten JJTree oder JTB, erreicht werden, die auf der Basis von Grammatiken für JavaCC Bäume erzeugen können.

Literatur. Zu JavaCC wird eine ausführliche Dokumentation mitgeliefert. Diese umfasst neben einem Benutzerhandbuch, diversen Tutorials, einer FAQ, einer kurzen Schnittstellenbeschreibung der erzeugten Lexer und Parser auch eine Dokumentation zu JJTree. Weitere Unterstützung für den Benutzer bietet ein Forum und eine Mailingliste. Ferner gibt es auf der Homepage Links zu weiteren (externen) Grammatiken.

Zu JavaCC gehört auch das Programm JJDoc, welches zu einer Grammatik eine HTML-Dokumentation erzeugen kann. Diese besteht aus einer BNF der Nichttermminale die untereinander verlinkt sind.

Verwendung. Bei JavaCC handelt es sich um ein Kommandozeilenwerkzeug. Als Aufrufparameter wird diesem, unter anderem, der Pfad zur Grammatik mitgeteilt, die in einen Parser übersetzt werden soll. Anschließend kann dieser kompiliert und (mit der zu parsenden Datei als Paramater) aufgerufen werden. Zur Laufzeit benötigen die erzeugten Parser keine Komponenten von JavaCC.

Eigene Erfahrungen. Da JavaCC in einem vorkompilierten JAR-Paket geliefert wird, entfällt die Einrichtung des Tools und man kann direkt damit arbeiten (ein JDK vorausgesetzt). Das Erzeugen der Parser aus den mitgebrachten Beispielgrammatiken (darunter auch eine für Java Version 5) funktionierte auf Anhieb wie dokumentiert. Allerdings musste zur Verwendung der Parser noch die CLASSPATH-Umgebungsvariable gesetzt werden, was die Dokumentation leider verschwieg. Danach war das Einbinden in eigene Projekte (auch Eclipse) mühelos möglich.

Position & Länge der Tokens im Quelltext. Aus der Schnittstellen-Beschreibung geht hervor, dass von jedem Token Anfang und Ende im Eingabestrom abgerufen werden kann. Behält man zu jedem Knoten im AST eine Referenz auf das Ursprungstoken, kann auch nach dem Parsing auf dieses zugegriffen werden.

Parsingtests. Die neuen Java 5 Ausdrücke bereiten dem Parser keine Schwierigkeiten, obschon seine Grammatik vom Februar 2004 stammt, also schon vor dem Erscheinen der endgültigen Fassung von Java 5. Allerdings stolpert der Parser in den fehlerfreien Quelltexten jedesmal über den Ausdruck aus Listing 2.

```
int[][] iaa = new int[10][10];
for (int ia[] : iaa)
        // ^ hier erwartet der Parser etwas anderes
```

Listing 2: Quelltextauszug der für JavaCC problematisch ist

Offensichtlich kommt dieser mit Arrays in for-Konstrukten nicht klar und erwartet eine Laufvariable zum Indexieren. Aus diesem Grund schlägt auch das Parsen von AllowedForEach-Construct.java, TestJSR201.java und TestJSR201Berichtigt.java fehl, da auch diese Klassen o. a. Code enthalten. Wird dieser Code auskommentiert, gelingt der Parsingvorgang erfolgreich.

Doch auch in der Klasse Test15 stolpert der Parser über den (eigentlich korrekten) Ausdruck aus Listing 3.

```
    new <String> K<Integer>("xh");
         ^ hier meldet der Parser einen Fehler
```

Listing 3: Quelltextauszug der für JavaCC problematisch ist

In diesem Fall kommt der Parser nicht mit Generic-Ausdrücken nach einem `new` klar. Wird der Code auskommentiert, meldet der Parser in der nächsten Zeile den selben Fehler bei einem ähnlichen Ausdruck. Nach weiterem Auskommentieren wird die Klasse endlich erfolgreich geparst.

Auch beim Parsen der fehlerhaften Quelltexte hinterlässt der Parser keinen guten Eindruck. Den simplen Fehler, dass mehr als ein Sichtbarkeits-Modifier bei einer Deklaration benutzt werden, erkennt der Parser nicht. Genauso kommentarlos verarbeitet der Parser die Modifier abstract und final in einer Deklaration. Wenigstens meldet der Parser einen Fehler bei einem falschen ForEachIn-Konstrukt.

Das Parsen der Quelltexte mit problematischen Newlines absolviert der Parser hingegen tadellos.

Eine Übersicht zu den Testergebnissen findet sich in Tabelle 7.

C.2.2 CUP

Eine kurze Übersicht zu den wichtigsten Daten von CUP findet sich in Tabelle 10.

Testquelltext	OK	Grund
AllowedGenericConstruct	Ja	Alles erkannt
AllowedForEachConstructs	Nein	Parse error at line 12, column 23. Encountered: :
AnotherAllowedForEachConstruct	Ja	Alles erkannt
Eric	Ja	Alles erkannt
EscapeSequences	Ja	Alles erkannt
Test15	Nein	Parse error at line 54, column 13. Encountered: <
TestJSR201Berichtigt	Nein	Parse error at line 22, column 23. Encountered: :
ForEachInShouldGiveParseError	Ja	Fehler erkannt
AbstractEnumShouldGiveParseError	Nein	Fehler nicht erkannt
AbstractFinalShouldGiveParseError	Nein	Fehler nicht erkannt
PublicPrivateShouldGiveParseError	Nein	Fehler nicht erkannt
TestsPositionsLinux	Ja	Alles erkannt
TestsPositionsMac	Ja	Alles erkannt
TestsPositionsWindows	Ja	Alles erkannt
TestZeilenumbrueche-ASCII	Ja	Alles erkannt

Tabelle 7: Ergebnisse Parsingtests JavaCC

Kriterium	Rechercheergebnis
Aktualität	neueste Version vom Januar 2006
Dokumentation	FAQ, Tutorials, Benutzerhandbuch
Support	Mailingliste, Forum
Parsergenerator in	Java
Parser in	Java
Grammatiken für	Java (bis 5)
Unterstützte OS	durch Java alle relevanten
Lizenz	Free BSD
Quelltexte	frei verfügbar

Tabelle 8: Übersicht JavaCC

Historie. Der Java(tm) Based Constructor of Useful Parsers (kurz CUP) wurde ursprünglich 1996 von Scott Hudson, Frank Flannery und C. Scott Ananian, an der Princeton University und Georgia Tech (siehe [24]), entwickelt. Die Quelltexte sind als Open Source frei verfügbar. Seit Ende 1999 wird CUP von der Technischen Universität München (siehe [25]) gepflegt und weiterentwickelt. Verantwortlich für die aktuellste Version (0.11a, Mai 2005) ist dort Michael Petter.

Charakterisierung. CUP ist ein javabasierter LR(1)-Parsergenerator, der ebenfalls javabasierte Parser erzeugt. Auf der Homepage von CUP sind, neben Grammatiken für eine Vielzahl von Sprachen, auch welche für die Java-Sprachversionen 1.0 bis 5, verfügbar. CUP erstellt aus den Grammatiken nur einen Parser. Scanner/Lexer kann dieses Werkzeug nicht erzeugen. Ein passender Scanner/Lexer kann wahlweise von Hand geschrieben oder mit anderen Scannergeneratoren erzeugt werden. Dazu bietet CUP ein Interface für javabasierte Scanner, unterstützt aber im Besonderen JLex und JFlex.

Auch im Falle von CUP geschieht die Baumerzeugung durch die generierten Java-Parser nicht automatisch. Diese Funktionalität muss, wie auch in JavaCC-Grammatiken, als Aktionscode in die Produktionen der Grammatik eingebaut werden.

Literatur. Die Dokumentation zu CUP ist leider nur sehr spärlich. Es wird lediglich ein kurzes Benutzerhandbuch mitgeliefert, welches die Erstellung von Grammatiken und die Erzeugung eines Parsers mit CUP behandelt. Auf der Homepage existiert weiterhin ein Merkblatt, welches die Erstellung von CUP mit Ant beschreibt. Darüberhinausgehende Fragen können per E-Mail an den Verantwortlichen gestellt werden.

Es bleibt dem Benutzer überlassen, herauszufinden wie die Grammatiken übersetzt und getestet werden können.

Verwendung. Auch CUP ist ein Kommandozeilenwerkzeug. Als Aufrufparameter wird diesem, unter anderem, der Pfad zur Grammatik mitgeteilt, die in einen Parser übersetzt werden soll. Der so erzeugte Parser muss dann in einem anderen Programm zusammen mit einem Lexer aufgerufen werden. Vorher muessen dem Lexer noch die zu parsenden Eingabedaten mitgegeben werden. Der Parsingvorgang wird dann durch den Parser, der zur Laufzeit weiterhin CUP benötigt, initiiert.

Eigene Erfahrung. Schon das Einrichten von CUP funktionierte nicht so reibungslos, wie es die Dokumentation suggerierte. Für Linuxsysteme wird ein Shellskript mitgeliefert, dass die Installation übernehmen soll. Leider brach es mit einer Fehlermeldung ab. Da CUP in unkompilierten Quelltexten geliefert wird, gelang uns die Kompilierung und das Ergänzen der CLASSPATH-Umgebungsvariablen schließlich von Hand.

Die Erzeugung eines Parsers aus einer Beispielgrammatik (einfacher Taschenrechner) gelang wie im Benutzerhandbuch beschrieben. Über die Benutzung dieses Parsers schweigt sich die Dokumentation allerdings aus. Wir konnten diesen zwar starten, fanden aber nicht heraus wie dieser zu bedienen ist. Einzig eine reproduzierbare Fehlermeldung ließ sich dem Programm entlocken (nach zweimaliger Eingabe eines arithmetischen Ausdrucks).

Auch die Erzeugung der Parser aus den Java-Grammatiken gelang zunächst erfolgreich. Allerdings ist auch in diesem Fall nicht beschrieben, wie diese Parser einzusetzen sind. Aus dem zu den Java-Grammatiken mitgelieferten Lexer und Codegerüst, welches eine Ausführung der Parser ermöglichen soll, konnten wir, nach vielem Ausprobieren und nochmaliger Erzeugung der Parser, herleiten wie diese zu benutzen sind. Nachdem wir in den Quelltexten zusätzlich noch einige imports ergänzt hatten, konnten wir die Parser endlich testen.

Danach gelang auch die Integration des Parsers in ein Eclipse Projekt. Da der Parser zur Laufzeit Komponenten von CUP benötigt, haben wir die Laufzeitumgebung von CUP in ein JAR gepackt und in das Eclipseprojekt importiert.

Position & Länge der Tokens im Quelltext. Zu diesem Thema schweigt sich die Dokumentation von CUP aus, ist dies doch eher Aufgabe des eingesetzten Lexers. Der mitgelieferte Beispiellexer bietet dafür aber keine Funktionaliäten.

Parsingtests. Das Parsen der fehlerfreien Quelltexte absolviert der Parser ohne Fehl und Tadel. Dies ist allerdings nicht weiter verwunderlich, da ein Teil der Testquelltexte aus CUP entnommen wurden.

Allerdings erkennt dieser keinen einzigen Fehler in den fehlerhaften Quelltexten. Auch ForEachInShouldGiveParseError.java verarbeitet der Parser ohne Kommentar. Der Grund dafür ist, dass die Javagrammatik für CUP nicht der finalen Java-Version 5 entstammt, sondern auf einer Preview basiert (genauer: Prototyp 2.2 von JSR-14 + JSR-201, vom Juli 2003).

Das Parsen der Quelltexte mit problematischen Newlines bereitet dem Parser hingegen keine Schwierigkeiten.

Eine Übersicht zu den Testergebnissen findet sich in Tabelle 9.

C.2.3 Coco/R

Eine kurze Übersicht zu den wichtigsten Daten von Coco/R findet sich in Tabelle 12.

Testquelltext	OK	Grund
AllowedGenericConstruct	Ja	Alles erkannt
AllowedForEachConstructs	Ja	Alles erkannt
AnotherAllowedForEachConstruct	Ja	Alles erkannt
Eric	Ja	Alles erkannt
EscapeSequences	Ja	Alles erkannt
Test15	Ja	Alles erkannt
TestJSR201Berichtigt	Ja	Alles erkannt
ForEachInShouldGiveParseError	Nein	Fehler nicht erkannt
AbstractEnumShouldGiveParseError	Neine	Fehler nicht erkannt
AbstractFinalShouldGiveParseError	Nein	Fehler nicht erkannt
PublicPrivateShouldGiveParseError	Nein	Fehler nicht erkannt
TestsPositionsLinux	Ja	Alles erkannt
TestsPositionsMac	Ja	Alles erkannt
TestsPositionsWindows	Ja	Alles erkannt
TestZeilenumbrueche-ASCII	Ja	Alles erkannt

Tabelle 9: Ergebnisse Parsingtests CUP

Kriterium	Rechercheergebnis
Aktualität	letztes Update Mai 2005
Dokumentation	Handbuch, Merkblatt
Support	über E-Mail Kontakt
Parsergenerator in	Java
Parser in	Java
Grammatiken für	Java (bis Prototyp 2.2 von JSR-14 + JSR-201)
Unterstützte OS	durch Java alle relevanten
Lizenz	GNU GPL
Quelltexte	frei verfügbar

Tabelle 10: Übersicht CUP

Historie. Der Coco/R Compilergenerator der Universität Linz ging aus einer Diplomarbeit aus dem Jahr 1983 hervor. Das damals von Hans Mössenböck geschriebene Werkzeug Coco wurde 1987 von ihm zu Coco/R (noch in Oberon) weiterentwickelt. Bis heute wurden, meist durch Dritte, Varianten dieses Werkzeug in eine Vielzahl von verschiedenen Programmiersprachen portiert. Die Quelltexte sind frei verfügbar und unterliegen der GNU GPL. Coco/R wird weiter gepflegt und erfuhr zuletzt im September 2005 ein Update.

Charakterisierung. Dieser LL(k)-Parsergenerator existiert in und für eine Vielzahl von Programmiersprachen. Einige Grammatiken sind auf der Homepage (siehe [26]) erhältlich - auch für Java, aber nur bis Sprachversion 1.4. Auch Coco/R bietet keine automatische Erzeugung von Bäumen. Diese Funktionalität muss, als Aktionscode zu den entsprechenden Produktionen, von Hand implementiert werden.

Literatur. Zu Coco/R existiert nur wenig Literatur. Auf der Homepage stehen ein recht ausführliches Handbuch zur Benutzung von Coco/R und einer kurzen Beschreibung der Schnittstellen der erzeugten Scanner und Parser, sowie zwei Artikel, die sich mit den Datenstrukturen des Parsergenerators und LL(1)-Konfliktauflösung beschäftigen. Ferner existiert eine Mailingliste, die über Updates von Coco/R informiert. Eine ausgewachsene API-Beschreibung, Tutorials oder ein Forum gibt es leider nicht.

Verwendung. Auch die Verwendung von Coco/R erfolgt über die Kommandozeile. Beim Aufruf muss als Parameter die Grammatik angegeben werden. Die so erzeugten Scanner und Parser müssen dann in einem eigenen Programm aufgerufen werden (benötigen aber keine Komponenten von Coco/R zur Laufzeit).

Zu Testzwecken können von der Homepage zu jeder Grammatik Testprogramme heruntergeladen werden.

Eigene Erfahrung. Coco/R kommt als JAR-Paket und kann, sofern ein JDK installiert ist, sofort benutzt werden. Die Erzeugung des Javaparsers funktioniert wie im Benutzerhandbuch angegeben. Zum Kompilieren muss sich dieser aber mit dem Testprogramm von der Homepage in einem package befinden, was die Dokumentation leider verschweigt. Nachdem wir diese Hürde genommen hatten, konnten wir ans Testen des Parsers gehen. Auch eine Integration des Parsers in eigenes Projekt war anschließend möglich.

Position & Länge der Tokens im Quelltext. Aus der Schnittstellenbeschreibung des Lexers (im Benutzerhandbuch) geht hervor, dass zu jedem Token der String (somit die Länge) und die Position im Quelltexte abgefragt werden kann. Auch hier müsste in jedem Knosten des AST eine Referenz zum Ursprungsknoten gehalten werden.

Parsingtests. Das Parsen der fehlerfreien Quelltexte führt zum Großteil zum erwarteten Ergebnis, da der Parser nicht mit Java 5 Konstrukten umgehen kann. Statische Imports, die neuen for-Ausdrücke und Generics sind ihm einfach nicht bekannt. Nur die Klasse Eric wird fehlerfrei geparst, da diese nur Ausdrücke enthält die schon in Java Version 1.4 erlaubt sind.

Beim Parsen der fehlerhaften Quelltexte erkennt der Parser alle Fehler bis auf den ungültigen „abstract final"-Ausdruck.

Die Quelltexten mit problematischen Newlines bereiten dem Parser überhaupt keine Schwierigkeiten.

Eine Übersicht zu den Testergebnissen findet sich in Tabelle 11.

Testquelltext	**OK**	**Grund**
AllowedGenericConstruct	Nein	line 4 col 37: lbrace expected
AllowedForEachConstructs	Nein	line 12 col 23: „;" expected
AnotherAllowedForEachConstruct	Nein	line 10 col 34: ident expected
Eric	Ja	Alles erkannt
EscapeSequences	Ja	Alles erkannt
Test15	Nein	line 5 col 20: lbrace expected
TestJSR201Berichtigt	Nein	line 1 col 8: ident expected
ForEachInShouldGiveParseError	Ja	line 23 col 13: lpar expected line 23 col 23: dot expected line 23 col 28: dot expected
AbstractEnumShouldGiveParseError	Nein	Fehler nicht erkannt
AbstractFinalShouldGiveParseError	Nein	Fehler nicht erkannt
PublicPrivateShouldGiveParseError	Ja	line 14 col 20: illegal combination of modifiers: public private
TestsPositionsLinux	Ja	Alles erkannt
TestsPositionsMac	Ja	Alles erkannt
TestsPositionsWindows	Ja	Alles erkannt
TestZeilenumbrueche-ASCII	Ja	Alles erkannt

Tabelle 11: Ergebnisse Parsingtests Coco/R

C.2.4 Cocktail

Eine kurze Übersicht zu den wichtigsten Daten von Cocktail findet sich in Tabelle 14.

Kriterium	Rechercheergebnis
Aktualität	September 2005
Dokumentation	Handbuch, Artikel
Support	über E-Mail-Kontakt
Parsergenerator in	Java, C++ , C#, Oberon, ua.
Parser in	Java, C++, C#
Grammatiken für	Java (bis 1.4), C#
Unterstützte OS	durch Java alle relevanten
Lizenz	GNU GPL
Quelltexte	frei verfügbar

Tabelle 12: Übersicht Coco/R

Historie. Die Entwicklung der ersten Version der Cocktail Toolbox begann 1988 an der GMD Forschungsstelle der Universität Karlsruhe. Nachdem die Forschungsstelle Ende 1993 geschlossen wurde, übernahm Josef Grosch die Rechte und gründete die Firma CoCoLab (siehe [27]). Die Software ist inzwischen kostenpflichtig und erhält in größeren aber regelmäßigen Abständen Updates.

Charakterisierung. Die Cocktail Toolbox beinhaltet eine Reihe von Werkzeugen für den Compilerbau, die Programmtransformation und die Programmanalyse. Ziel bei der Entwicklung dieser Werkzeuge war eine gesteigerter Umfang und eine gesteigerte Leistungsfähigkeit im Vergleich zu Lex / Yacc.

In der Toolbox sind u.a. Generatoren zur Erzeugung von LL(1)-, LR(1)- und LALR(2)-Parsern, sowie auch ein AST-Generator und -Transformator enthalten. Dabei unterstützen die Tools (neben anderen Sprachen wie Cobol, PL/I, SQL, C, C++, C# und VB) auch Java bis einschließlich Version 5 und sind unter einer graphischen Benutzeroberfläche zusammengefasst.

Verwendung. Da die Nutzung kostenpflichtig ist, steht auf der Homepage nur eine beschränkt lauffähige Version zur Verfügung. Diese kommt ohne Oberfläche und besteht im Wesentlichen aus einer Sammlung der Kommandozeilenwerkzeuge. Da aber auch keine Grammatiken dabei sind, entfällt der Test der Werkzeuge.

Allerdings werden auf der Homepage auch diverse Demoparser für eine Reihe von Sprachen zum Download angeboten. Im Weiteren haben wir den Java 5 Parser genauer untersucht.

Literatur. Auf der Homepage sind, neben ausführlichen Benutzerhandbüchern zu allen Werkzeugen, auch FAQs, Tutorials und Forschungsveröffentlichungen verfügbar. Eine API-Beschreibung, ein Forum oder eine Mailingliste gibt es nicht.

Eigene Erfahrungen. Da der Demoparser mit einer Benutzeroberfläche ausgestattet ist, fällt dessen Benutzung denkbar einfach aus. Nach dem Öffnen einer Datei zeigt dieser den AST des Quelltextes an und meldet eventuelle Parsingfehler.

Position & Länge der Tokens im Quelltext. Zu diesem Thema waren in der Dokumentation keine Informationen zu finden. In dem Demoprogramm wird aber ersichtlich, dass die Position (als Zeile und Spalte) in den Knoten des AST aufgenommen wird.

Parsingtests. Das Parsen der fehlerfreien Quelltexte geschieht fast problemlos. Einzig die Klasse Test15 bereitet dem Parser Schwierigkeiten - teilweise führte dies sogar zum Absturz des Programms.

Beim Parsen der fehlerhaften Quelltexte findet der Parser nur den nicht mehr erlaubten, `for each ( .. in .. )`-Ausdruck. Die abstrakte Enum-Konstruktion und ungültige Modifierkombinationen bemerkt der Parser nicht.

Die Parsingtests der Quelltexte mit problematischen Newlines verliefen hingegen positiv.

Eine Übersicht zu den Testergebnissen findet sich in Tabelle 13.

Testquelltext	OK	Grund
AllowedGenericConstruct	Ja	Alles erkannt
AllowedForEachConstructs	Ja	Alles erkannt
AnotherAllowedForEachConstruct	Ja	Alles erkannt
Eric	Ja	Alles erkannt
EscapeSequences	Ja	Alles erkannt
Test15	Nein	Führt teilweise zum Absturz
TestJSR201Berichtigt	Ja	Alles erkannt
ForEachInShouldGiveParseError	Ja	23, 13 Error found/expected : Identifier/(25, 17 Error found/expected : Identifier/(
AbstractEnumShouldGiveParseError	Nein	Fehler nicht erkannt
AbstractFinalShouldGiveParseError	Nein	Fehler nicht erkannt
PublicPrivateShouldGiveParseError	Nein	Fehler nicht erkannt
TestsPositionsLinux	Ja	Alles erkannt
TestsPositionsMac	Ja	Alles erkannt
TestsPositionsWindows	Ja	Alles erkannt
TestZeilenumbrueche-ASCII	Ja	Alles erkannt

Tabelle 13: Ergebnisse Parsingtests Cocktail

Kriterium	Rechercheergebnis
Aktualität	letztes Update im August 2005
Dokumentation	Einführung, Tutorials, FAQ, Handbücher, Artikel
Support	über E-Mail
Parsergenerator in	C
Parser in	Java, C++, C, Ada, Eiffel
Grammatiken für	Java (bis 5), C#, C++, C, uva.
Unterstützte OS	Linux, Win32
Lizenz	kostenpflichtig
Quelltexte	nicht verfügbar

Tabelle 14: Übersicht Cocktail

C.2.5 ANTLR

Eine kurze Übersicht zu den wichtigsten Daten von ANTLR findet sich in Tabelle 17.

Historie. Die Entwicklung des „ANother Tool for Language Recognition“ begann 1988, ursprünglich als Projekt an der Purdue Universität (West Lafayette, Indiana, USA), damals noch unter dem Namen „YUCC“ bzw. „PCCTS“. Dieses Projekt wurde 1989 Inhalt der Masterarbeit von Terence Parr, welcher fortan die Schlüsselfigur hinter ANTLR wurde.

Die erste öffentlich verfügbare Version erschien 1990 und ist seit jeher quelloffen verfügbar. Das Projekt wurde kontinuierlich weiterentwickelt und Terence Parr beschäftigt sich heute noch, als Professor an der Universität von San Francisco, mit ANTLR. Die derzeit aktuelle Version ist 2.7.6. Allerdings sind die Entwicklungsarbeiten an Version 3 schon weit fortgeschritten und die Veröffentlichung liegt in naher Zukunft.

Charakterisierung. ANTLR ist ein Framework zur Erstellung von Compilern bzw. deren einzelner Bestandteile, mit besonderem Schwerpunkt auf der Unterstützung von Bäumen (Erzeugung, Traversierung, Konvertierung). ANTLR liegt selbst in verschiedenen Sprachen vor und kann LL(*)-Parser in Java, C++ und C# erzeugen. Dabei unterstützt ANTLR die automatische Erzeugung von ASTs. Deren Traversierung kann in separaten Baumgrammatiken beschrieben werden.

Ferner soll in Version 3 noch eine graphische Benutzeroberfläche (ANTLRWorks) zum Entwickeln, Debuggen und Testen von Grammatiken hinzukommen.

Für Java 1.5 sind auf der Homepage derzeit drei verschiedene Grammatiken verfügbar. Desweiteren sind Grammatiken für eine Vielzahl weiterer Sprachen dort abrufbar.

Literatur. Auf der ANTLR Homepage (siehe [13]) steht eine umfangreiche Dokumentation, bestehend aus einer langen FAQ, Tutorials, einem Benutzerhandbuch, einer API-Beschreibung des Frameworks und zusätzlichen Veröffentlichungen von, teils wissenschaftlichen, Arbeiten zum Thema Compilerbau zur Verfügung. Fragen, die darüber hinausgehen, können in Foren sowie in einer Mailingliste gestellt werden.

Verwendung. Bei allen im Framework enthaltenen Tools handelt es sich um Kommandozeilenprogramme. Zusätzlich existiert eine graphische Oberfläche für die Benutzung der Kommandozeilenwerkzeuge unter Eclipse, ANTLR Studio genannt. Allerdings stammt es von einem Dritten und ist kostenpflichtig. Deshalb gehen wir im Weiteren nur auf die Kommandozeilenverwendung ein.

Zur Erzeugung der Lexer und Parser muss ANTLR, mit einer Grammatik als Parameter, aufgerufen werden. Die so erzeugten Lexer und Parser müssen dann zur Verwendung in einem separaten Programm eingebaut werden. Dabei benötigen diese weiterhin ANTLR zur Laufzeit.

Eigene Erfahrungen. ANTLR wird (in der Javaversion) wahlweise in einem vorkompilierten JAR-Paket oder als Quelltextversion geliefert. Die Installation durch ein Setup-Programm verlief fehlerfrei und wir konnten direkt mit der Erzeugung der Parser beginnen. Rund um dieses Projekt existiert offensichtlich eine vergleichsweise große und aktive Community. Dieser entstammen auch die drei Grammatiken für Java 1.5, welche wir im weiteren gestestet haben.

Die Erzeugung und Kompilierung des jeweiligen Lexers und Parsers aus den beiden Javagrammatiken von Michael Stahl und Michael Studman verlief fehlerfrei, so dass wir mit einem, auf der Homepage verfügbaren, Beispielprogramm die Parsingtests durchführen konnten.

Die dritte verfügbare Grammatik von Scott Wisniewski baut auf der von Studman auf. Sie soll die volle Spanne der Java-Identifier erkennen und mit Unicode-Escapesequenzen umgehen können. Für die Unicode-Unterstützung wird ein in C# (für .NET 2.0) geschriebener Präprozessor mitgeliefert. Um diesen benutzen zu können, mussten wir zunächst einmal ein kleines Kommandozeilenprogramm schreiben, welches die Präprozessor-Klasse benutzt und die Eingabedateien verarbeitet. Der Lexer und der Parser ließen sich zwar erzeugen, jedoch ließ sich ersterer nicht kompilieren, da an zwei Stellen im Javaquelltext „Char.IsLetter(...)“ bzw. „Char.IsLetterOrNumerical(...)“ erzeugt wurden. Nach manuellem Ersetzen durch „Character.isLetter(...)“ bzw. „Character.isLetterOrNumerical(...)“ (siehe API-Beschreibung von Java 5) war eine Kompilierung möglich. Allerdings konnten wir den Parser nicht testen, da dieser an den von uns korrigierten Stellen abbrach.

Position & Länge der Tokens im Quelltext. Laut API-Beschreibung lässt sich die Position des Tokens im Quelltext nicht direkt abrufen. Lediglich Zeile, Spalte und Länge (über die Länge des Strings) des Tokens sind einsehbar.

Es wäre aber möglich, die Klasse Token abzuleiten und dort die Position einzubauen. Dazu müsste allerdings noch der Lexer angepasst werden, damit dieser die Position auch in das Token einträgt.

Die Knoten des AST (von ANTLR) enthalten eine Referenz auf das Token. Somit ist ein Abruf der benötigten Eigenschaften auch nach dem Parsing möglich.

Parsingtests - Parser nach Grammatik von Michael Stahl. Alle Testquelltexte mit fehlerfreien Konstrukten wurden von diesem Parser akzeptiert, ausgenommen diejenigen, welche `enum`-Konstrukte enthalten. Offensichtlich ist die vorhandene Grammatik nicht vollständig und erkennt diesen Ausdruck nicht.

Das ungültige `for each ( .. in ..)` wurde korrekterweise nicht akzeptiert, alle ungültigen und widersprüchlichen Modifier wurden allerdings fälschlicherweise akzeptiert.

Die Quelltexte mit problematischen Zeilenumbrüchen wurden fehlerfrei erkannt.

Eine Übersicht zu den Testergebnissen findet sich in Tabelle 15.

Parsingtests - Parser nach Grammatik von Michael Studman. Alle Testquelltexte mit fehlerfreien Konstrukten wurden von diesem Parser akzeptiert, im Gegensatz zur Grammatik von Stahl auch solche mit `enum`-Ausdrücken.

Die restlichen Ergebnisse sind identisch: Das ungültige `for each ( .. in .. )` wurde korrekterweise nicht akzeptiert, alle ungültigen und widersprüchlichen Modifier wurden fälschlicherweise akzeptiert.

Die Quelltexte mit problematischen Zeilenumbrüchen wurden fehlerfrei erkannt.

Eine Übersicht zu den Testergebnissen findet sich in Tabelle 16.

C.3 Werkzeuge, die Parsergeneratoren benutzen

In diesem Abschnitt werden Tools untersucht, welche auf einen Parsergenerator aufbauen oder mit einem durch einen Generator erzeugten Parser arbeiten und dafür eigene Java-Grammatiken mitbringen.

Testquelltext	OK	Grund
AllowedGenericConstruct	Ja	Alles erkannt
AllowedForEachConstructs	Ja	Alles erkannt
AnotherAllowedForEachConstruct	Ja	Alles erkannt
Eric	Ja	Alles erkannt
EscapeSequences	Ja	Alles erkannt
Test15	Ja	Alles erkannt
TestJSR201Berichtigt	Nein	Erkennt offensichtlich das Keyword „enum“ nicht
ForEachInShouldGiveParseError	Ja	Erwartet korrekterweise „(„ anstelle von „each“
AbstractEnumShouldGiveParseError	Nein	Erkennt nicht den eigentlichen Fehler, sondern erkennt das „enum“ nicht
AbstractFinalShouldGiveParseError	Nein	Beide widersprüchlichen Modifier werden ohne Fehlermeldung in den AST aufgenommen
PublicPrivateShouldGiveParseError	Nein	Beide widersprüchlichen Modifier werden ohne Fehlermeldung in den AST aufgenommen
TestsPositionsLinux	Ja	Alles erkannt
TestsPositionsMac	Ja	Alles erkannt
TestsPositionsWindows	Ja	Alles erkannt
TestZeilenumbrueche-ASCII	Ja	Alles erkannt

Tabelle 15: Ergebnisse Parsingtests ANTLR (Parser nach Stahl-Grammatik)

Testquelltext	OK	Grund
AllowedGenericConstruct	Ja	Alles erkannt
AllowedForEachConstructs	Ja	Alles erkannt
AnotherAllowedForEachConstruct	Ja	Alles erkannt
Eric	Ja	Alles erkannt
EscapeSequences	Ja	Alles erkannt
Test15	Ja	Alles erkannt
TestJSR201Berichtigt	Ja	Alles erkannt
ForEachInShouldGiveParseError	Ja	Erwartet korrekterweise „(„ anstelle von „each“
AbstractEnumShouldGiveParseError	Nein	Der ungültige Modifier wird ohne Fehlermeldung in den AST aufgenommen
AbstractFinalShouldGiveParseError	Nein	Beide widersprüchlichen Modifier werden ohne Fehlermeldung in den AST aufgenommen
PublicPrivateShouldGiveParseError	Nein	Beide widersprüchlichen Modifier werden ohne Fehlermeldung in den AST aufgenommen
TestsPositionsLinux	Ja	Alles erkannt
TestsPositionsMac	Ja	Alles erkannt
TestsPositionsWindows	Ja	Alles erkannt
TestZeilenumbrueche-ASCII	Ja	Alles erkannt

Tabelle 16: Ergebnisse Parsingtests ANTLR (Parser nach Studman-Grammatik)

Kriterium	Rechercheergebnis
Aktualität	neue Version in Entwicklung, Benutzer tragen immer wieder Grammatiken bei
Dokumentation	Einführung, Präsentationen, Tutorials, FAQ, Handbuch, Artikel, API-Spezifikation, Wiki in Arbeit
Support	Mailingliste, E-Mail
Parsergenerator in	Java, C++
Parser in	Java, C++, C#, Python (in Kürze)
Grammatiken für	Java (bis 5), C#, C++, C, uva.
Unterstützte OS	durch Java alle relevanten
Lizenz	Public Domain (v2), Free BSD (v3)
Quelltexte	frei verfügbar

Tabelle 17: Übersicht ANTLR

C.3.1 JAbstract - A full Abstract Syntax and Parser for Java

Eine kurze Übersicht zu den wichtigsten Daten von JAbstract findet sich in Tabelle 18.

Historie. Die Forschungsgruppe um Sebastian Danicic an der Goldsmiths Universität London beschäftigt sich unter anderem mit Programmtransformationen. Daraus ging auch JAbstract hervor. Mittlerweile wird dieses Projekt nicht mehr weiterentwickelt, da sich die Forschungsgruppe anderen Ansätzen zugewandt hat.

Charakterisierung. JAbstract ist ein Framework, welches Javaklassen zur Programmtransformation mittels eines AST bereitstellt. Um den AST (in dem selbst noch die Kommentare enthalten sind) aufzubauen, nutzt JAbstract eine JavaCC Grammatik, die aber nur Java bis Sprachversion 1.1 verarbeiten kann.

Literatur. Die Dokumentation zu diesem Projekt ist leider sehr spärlich. Auf der Homepage (siehe [28]) kann man nur die Quelltexte und die zugehörige, mit Javadoc erzeugte API-Beschreibung einsehen.

Verwendung. Der Parser von JAbstract wurde schon erzeugt, lediglich die Quelltexte müssen noch kompiliert werden. Danach kann der Parser über die Kommandozeile aufgerufen werden. Als Parameter gibt man ihm die zu parsende Klasse mit.

Eigene Erfahrungen. Liest man die wenigen Zeilen Beschreibung auf der Homepage zu JAbstract, zwingt sich einem der Gedanke auf, dass es sich hier nur um ein Proof-of-Concept-Projekt handelt. Auf eine E-Mail von uns, ob JAbstract noch gepflegt werde, bekamen wir die Antwort, dass wir uns lieber das System Stratego anschauen sollten, da dieses wesentlich besser für unsere Zwecke geeignet sei (Stratego erzeugt aber keine Parser in Java und wurde deshalb nicht weiter untersucht).

Position & Länge der Token im Quelltext. Da es sich um eine JavaCC-Grammatik handelt, kann die Position der Tokens und deren Länge abgerufen werden. Wird eine Referenz im AST auf das Ursprungstoken gehalten, können Position und Länge auch nach dem Parsing abgerufen werden.

Parsingtest. Diese entfallen, da nur die Java-Sprachversion 1.1 unterstützt wird.

Kriterium	Rechercheergebnis
Aktualität	letztes Update Januar 2005
Dokumentation	API-Spezifikation
Support	über E-Mail Kontakt
Parsergenerator in	Java (javacc)
Parser in	Java
Grammatiken für	Java (bis 1.1)
Unterstützte OS	durch Java alle relevanten
Lizenz	GNU GPL
Quelltexte	frei verfügbar

Tabelle 18: Übersicht JAbstract

C.3.2 JRefactory

Eine kurze Übersicht zu den wichtigsten Daten von JRefactory findet sich in Tabelle 19.

Historie. Die Entwicklung von JRefactory wurde 1999 von Chris Seguin begonnen und 2002 von Mike Atkinson übernommen. Die letzte veröffentlichte Version stammt von Ende 2003 und auch die Homepage ist seit diesem Zeitpunkt nicht mehr wesentlich aktualisiert worden.

Mittlerweile kann das Projekt als eingestellt angesehen werden, da seit 2004 weder Aktivitäten in der Entwicklung, noch in der begleitenden Mailingliste, dem Forum oder der Homepage verzeichnet sind.

Charakterisierung. Bei JRefactory handelt es sich um ein Tool, welches Refactorings von Javaquelltexten unterstützt. Es kann als reines Kommandozeilenwerkzeug benutzt werden, bringt aber auch eine eigene Oberfläche mit. Ferner existieren PlugIns für diverse IDEs die Java unterstützen.

Interressant für uns sind allerdigs nicht die Refactoring-Funktionen, sondern der intergrierte Javaparser. Dieser wird als separates Paket zum Download angeboten. Dabei handelt es sich um einen Parser, der auf einer eigenen JavaCC-Grammatik basiert. Diese soll auch Sprachkonstrukte aus Java Version 5 kennen und einen AST erzeugen. So behandeln wir im Rest des Abschnitts nur den Javaparser genauer.

Literatur. Zu JRefactory existiert keine wirklich brauchbare Dokumentation. Die Homepage (siehe [29]) beschränkt sich eigentlich nur auf eine Beschreibung der Features.

Verwendung. Aufgrund der o. a. Dokumentationslage zogen wir die Dokumentation von JavaCC hinzu und konnten feststellen, dass der Parser in ein separates Programm eingebunden werden muss, welches die Vorbehandlung der Eingabedaten und das Anstoßen des Parsings übernimmt.

Eigene Erfahrungen. Da der Parser in einem vorkompilierten JAR-Paket geliefert wird, entfällt die Erzeugung mit JavaCC. Es wird zwar explizit darauf hingewiesen, dass Informationen zur Benutzung auf der Homepage verfügbar seien - leider ist dies nicht der Fall.

Es gelang uns zwar eine Javaklasse zu schreiben, die einen Javaquelltext an den Parser weiterreicht und den Parsingvorgang (wie in der JavaCC-Dokumentation beschrieben) anstösst. Allerdings bekamen wir vom Parser keine Rückmeldung. So ist uns nicht klar, ob der Parsingvorgang überhaupt begann und ob dieser erfolgreich war.

Parsingtests. Die zugrundeliegende Grammatik stammt vom Juni 2003, ist also älteren Datums als die endgültige Fassung von Java 5. Unsere bisherigen Erfahrungen haben jedoch gezeigt, dass dies nicht von Vorteil für die Erkennungsrate der Parser ist. Leider mussten die Parsingtests aber entfallen, da es uns nicht gelang dem Parser eine brauchbare Rückmeldung zu entlocken.

Kriterium	Rechercheergebnis
Aktualität	letztes Update November 2003
Dokumentation	keine
Support	Mailingliste, Forum
Parsergenerator in	Java (da JavaCC)
Parser in	Java
Grammatiken für	Java (bis 5) für JavaCC
Unterstützte OS	durch Java alle relevanten
Lizenz	GNU GPL
Quelltexte	frei verfügbar

Tabelle 19: Übersicht JRefactory

C.3.3 FUJABA

Eine kurze Übersicht zu den wichtigsten Daten von FUJABA findet sich in Tabelle 20.

Historie. FUJABA steht für „From UML to Java and back again“ und wurde 1997 von Tomas Klein, Ulrich Nickel, Jörg Niere und Albert Zündorf an der Universität Paderborn entwickelt. Seitdem unterliegt es einer ständigen Weiterentwicklung. Die aktuelle Version stammt vom März 2005, allerdings ist für März 2006 schon die nächste Version angekündigt.

FUJABA ist Public Domain und die Quelltexte sind frei verfügbar.

Charakterisierung. Die FUJABA Tool Suite stellt Werkzeuge zur Verfügung, die aus UML Diagrammen einen javabasierten Prototypen erstellen. Der umgekehrte Weg ist ebenfalls möglich, sprich Java Quelltexte zu parsen und (bis zu einem gewissen Grad) in UML Diagramme umsetzen zu können. Dabei wird auch die Verwendung von Patterns durch eine eigene Spezifikationssprache unterstützt.

Für uns ist allerdings lediglich der als PlugIn realisierte Javaparser von Bedeutung. Bisher kann dieser nur Javaquelltexte bis Java 1.4 parsen und den dazugehörigen AST aufbauen. Parsing von Java-5-Quelltexten ist geplant, aber noch nicht in Arbeit.

Literatur. Auf der Homepage (siehe [30]) von FUJABA existiert eine ausführliche Dokumentation zu dessen Verwendung, für Entwickler existieren hingegen nur eine API-Beschreibung von FUJABA und ein paar Einträge im Wiki. Bei Fragen kann man sich an eine Mailingliste wenden.

Verwendung. Der Parser muss in ein separates Programm eingebunden werden, welches die Bereitstellung der Eingabedaten und das Anstoßen des Parsings übernimmt.

Eigene Erfahrungen. Durch Ausprobieren und Entpacken der PlugIn-Packages von Fujaba haben wir herausgefunden, dass sich FUJABA einer freizugänglichen JavaCC-Grammatik bedient. Aus dieser wurde erst mit jtb (Java Tree Builder, siehe [31]) eine weitere Grammatik und Klassen für einen AST erzeugt. Aus der neuen Grammatik wurde schließlich mit JavaCC der Parser generiert, der den AST aufbaut.

Darüber hinaus existiert ein FUJABA PlugIn, welches JavaAST heisst und auf dem AST arbeitet. Zwar kann man sich die Quelltexte von der Homepage herunterladen, doch leider sind gerade die Quelltexte dieses PlugIns nicht frei zugänglich. Auf Nachfragen in der Mailinglist wurde uns mitgeteilt, dass ein Zugriff auf den CVS-Server zwar grundsätzlich möglich sei, allerdings ist auch hier dieses PlugIn ausgeschlossen, da es im Moment Gegenstand der Forschung sei.

Es gelang uns zwar den Parser zu erzeugen, aber nicht zu kompilieren. Auch die Verwendung des bereits kompilierten Parsers aus den FUJABA Binaries gelang uns nicht.

Auf eine weitere Anfrage hin erfuhren wir, dass der erzeugte AST nicht vollständig ist, da dieser lediglich die Methodenrümpfe enthält sowie in der Darstellung der Ausdrücke stark vereinfacht ist. Alle weiteren Informationen wie etwa die Methodenköpfe oder Klassen werden nicht im AST, sondern im verwendeten UML-Modell gespeichert, so dass eine weitere Betrachtung und weitere Tests für dieses Tool entfallen.

Position & Länge der Token im Quelltext. Da es sich um eine JavaCC-Grammatik handelt, gilt auch hier alles was bereits im Abschnitt JavaCC geschrieben wurde.

Parsingtest. Entfallen, da Parser nicht zu benutzen war.

Kriterium	Rechercheergebnis
Aktualität	neue Version in Entwicklung (v5)
Dokumentation	Wiki, API-Spezifikation, FAQ, Tutorials
Support	Mailingliste, Forum
Parsergenerator in	Java (da JavaCC)
Parser in	Java
Grammatiken für	Java (bis 1.4) für JavaCC
Unterstützte OS	durch Java alle relevanten
Lizenz	teilweise open source
Quelltexte	teilweise frei verfügbar

Tabelle 20: Übersicht FUJABA

C.4 Quelloffene Java-Compiler Implementationen

Da auch beim Kompilieren ein Parsing notwendig ist, haben wir die nachfolgenden quelloffenen Java-Compiler hinsichtlich der verwendeten Parser bzw. Grammatiken untersucht. Der Originalcompiler von Sun ist selbst leider nicht quelloffen (im Gegensatz zur Klassenbibliothek des JDK) und schied damit aus.

C.4.1 GCJ

Eine kurze Übersicht zu den wichtigsten Daten von GCJ findet sich in Tabelle 21.

Historie. Die erste Version des GCC wurde im März 1987 veröffentlicht, damals allerdings noch als reiner C-Compiler. Mit der Zeit wurden Compiler für weitere Sprachen integriert, und der GNU C Compiler zur GNU Compiler Collection (beides als GCC bezeichnet) umbenannt. Mit der Version 2.95, welche im Juli 1999 veröffentlicht wurde, wurde erstmals ein Java-Compiler integriert. Mittlerweile ist die GCC bei Version 3.4.5 (Veröffentlichung vom 30.11.2005) angelangt und die derzeitige Entwicklung konzentriert sich auf die Version 4, von welcher bereits Entwicklerversionen verfügbar sind.

Charakterisierung. Der GNU Compiler for the Java(tm) Programming Language (GCJ; siehe [32]) ist Teil der GNU Compiler Collection, welche neben dem GCJ auch Compiler für C, C++, Objective-C, Fortran und Ada, sowie Bibliotheken für die genannten Sprachen beinhaltet. Die komplette GCC ist quelloffen unter der GNU Public License (GPL) verfügbar.

Der GCJ selbst ist in C geschrieben und benutzt einen mit Bison erzeugten Parser. Bison ist ein Parsergenerator, welcher kontextfreie LALR-Parser in C erzeugt. Bison selbst ist ebenfalls Teil des GNU-Projektes.

Verwendung. Bei den Programmen des GCC-Projektes handelt es sich um Kommandozeilenprogramme, so auch beim GJC und bei Bison.

Kriterium	Rechercheergebnis
Aktualität	letztes Update Oktober 2005
Dokumentation	Wiki, Handbuch, FAQ, API-Spezifikation
Support	Mailingliste
Parsergenerator in	C (da Bison)
Parser in	C
Grammatiken für	Java (Version ?)
Unterstützte OS	Linux, Windows
Lizenz	GPL
Quelltexte	frei verfügbar

Tabelle 21: Übersicht GCJ

C.4.2 Java Espresso

Eine kurze Übersicht zu den wichtigsten Daten von ANTLR findet sich in Tabelle 22.

Die Arbeitsweise dieses Compilers ist ausführlich auf der Homepage (siehe [33]) dokumentiert. Dieser baut beim Parsen einen AST auf. Danach wird das Type-Checking durchgeführt,

indem von jedem Knoten die Methode typeCheck() aufgerufen wird (die Knoten sind in Klassen beschrieben). Um dabei beispielsweise die Sichtbarkeiten aufzulösen, bedient sich Espresso einer eigenen Typhierarchie.

Kriterium	Rechercheergebnis
Aktualität	letztes Update August 1998
Dokumentation	Arbeitsweise, API-Spezifikation
Support	über E-Mail
Parsergenerator in	Java (da JavaCC)
Parser in	Java
Grammatiken für	Java (nur 1.0) für JavaCC
Unterstützte OS	durch Java alle relevanten
Lizenz	open source
Quelltexte	frei verfügbar

Tabelle 22: Übersicht Java Espresso

C.5 Eclipse

Historie. Die hinter Eclipse stehende Organisation wurde im November 2001 als Gemeinschaftsprojekt mehrerer großer IT-Firmen gegründet (Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft, Webgain). Seitdem ist die Zahl der Mitgliedsfirmen erheblich gewachsen und die Organisation wurde zwischenzeitlich zu einer gemeinnützigen („non-profit“) Organisation umfunktioniert (siehe [34]). Vor diesem Hintergrund ist die Aktualität der Versionen (aktuell: 3.1.2 vom 18.01.2006) selbstverständlich und die weitere Fortentwicklung kann als gesichert gelten, wenn man zusätzlich die inzwischen weite Verbreitung von Eclipse bedenkt. Die Dokumentation ist bei allen Teilen der Software in größerem Maße vorhanden.

Charakterisierung. Bei Eclipse handelt es sich um eine frei verfügbare graphische Entwicklungsumgebung. Das Programm ist in Java implementiert und OpenSource-Software.

C.5.1 EMF - Eclipse Modeling Framework

Charakterisierung. Beim Eclipse Modeling Framework handelt es sich um die Architektur hinter Eclipse. Grundsätzlich stellt Eclipse lediglich die grundlegenden Funktionen einer Entwicklungsumgebung bereit und ist nicht auf eine bestimmte Programmiersprache festgelegt. Die Funktionalität zum Entwickeln in bestimmten Programmiersprachen wird dabei über

Plugins realisiert, derzeit ist eine offizielle Unterstützung für COBOL, Fortran, C, C++, Java und AspectJ vorhanden (sowie inoffiziell noch einige weitere, wie z.B. CaesarJ). Das EMF an sich bietet keinerlei Modellierungsstruktur, welche feingranularer als die Klassenebene ist und ist folglich für den geplanten Java-Extraktor völlig ungeeignet. Informationen zur genaueren Struktur der Konzepte von Eclipse, EMF und Plugins finden sich u.a. in [10].

C.5.2 JDT Core

Eine kurze Übersicht zu den wichtigsten Daten von JDT Code findet sich in Tabelle 23.

Historie. Die erste Version der JDT wurde im Juli 2002 veröffentlicht und seitdem bis auf die aktuelle Version 3.1.2 vom 18.01.2006 (entspricht in Version und Datum dem Eclipse-Paket selbst) weiterentwickelt, wobei bereits die Arbeiten an der nächsten Version 3.2.x im Gange sind.

Charakterisierung. Das Paket Java Developer Tools (siehe [35]) beinhaltet die Komponenten zur Java-Entwicklung unter Ecplise und ist seinerseits in das Core-, Debug- und User Interface-Paket unterteilt. Dabei beinhaltet das Core-Paket u.a. Funktionalitäten zur Repräsentation von Javacode als AST, inklusive Grammatik und Parser. Seit Version 3.1.0 unterstützt JDT Core auch Java5.

In [10] werden Funktionalitäten des Refactorings auf dem AST des JDT aufgebaut. Dabei sind Modifikationen im AST möglich, welche dann wieder auf den zugrunde liegenden Code übertragbar sind. Dazu sind Positionsangaben der einzelnen Codelemente im AST gespeichert (anders wäre diese Funktionalität wohl kaum realisierbar).

Aufgrund der Auslegung der JDT hinsichtlich der reinen Verwendung als Eclipse-Plugin ist es ohne größeren Aufwand (und vor allem ohne Eingriff in den Code der JDT selbst) nicht möglich, einen darauf aufbauenden Extraktor für Java zu implementieren, welcher ohne Eclipse und als Kommandozeilen-Programm eigenständig lauffähig wäre. Einer Implementation des Extraktors seinerseits als Eclipse-Plugin stünden jedoch keinerlei technischen Hürden im Weg.

Verwendung. Die Funktionalitäten von JDT Core sind nicht direkt als eigenes Programm zugänglich, sondern werden durch das Paket „JDT User Interface“ in die graphische Oberfläche von Eclipse integriert. Ebenso ist es möglich, die Funktionalität über die Plugin-Schnittstellen in eigenen Eclipse-Plugins zu verwenden.

Kriterium	Rechercheergebnis
Aktualität	letztes Update (Release-Version) 03.10.2005
Dokumentation	Handbuch, API-Spezifikation, Newsgroups, FAQ, Forum, etc.
Support	über E-Mail, Newsgroups, Forum, IRC
Parsergenerator in	- entfällt -
Parser in	Java
Grammatiken für	Java bis 1.5
Unterstützte OS	durch Java alle relevanten
Lizenz	open source
Quelltexte	frei verfügbar

Tabelle 23: Übersicht JDTCore / Eclipse

C.6 Fazit

Nach der vorangegangenen Übersicht können nun einige Tools aus der engeren Auswahl ausgeschlossen werden.

JAbstract bietet zwar die Möglichkeit zur Erzeugung eines AST, entfällt aber aufgrund der völlig veralteten Grammatik und der mangelnden Weiterentwicklung.

JRefactory unterstützt zwar Parsing von Java 1.5, ist aber, aufgrund der mangelhaften Dokumentation des Parsers (welche schon die Parsing-Tests vereitelte) und der Inaktivität der Entwickler seit fast zwei Jahren, als stabile Grundlage ebenfalls ausgeschlossen.

Fujaba wird aktiv weiterentwickelt, liefert aber einen AST, der bei weitem nicht vollständig ist. Desweiteren wird derzeit lediglich Java bis Version 1.4 unterstützt, so dass dieses Tool technisch ungeeignet ist.

GCJ entfällt aufgrund der Tatsache, dass sowohl der Parsergenerator als auch der erzeugte Parser in C vorliegen.

Eclipse bzw. das JDT-Core-Paket besitzen zwar weitgehend alle benötigten Funktionen, implizieren aber eine Implementierung des Faktenextraktors als Eclipse-Plugin, so dass eine Integration als Kommandozeilenwerkzeug in Gupro entfällt.

Cocktail entfällt aufgrund der Tatsache, dass es kostenpflichtig ist und mit den verfügbaren Demoversionen keine genaue Evaluierung der Funktionalität möglich ist.

Coco/R unterstützt derzeit nur Java bis Version 1.4, ist nur begrenzt dokumentiert und bietet keine automatische Erzeugung eines AST an.

Somit verbleiben noch JavaCC, CUP und ANTLR in der engeren Auswahl. JavaCC ist sehr gut dokumentiert und besitzt eine gute Community, bietet aber ebenfalls keine automatische

Erzeugung des AST an. Darüber hinaus ist die derzeit verfügbare Grammatik für Java 1.5 fehlerhaft.

CUP erzeugt ebenfalls keinen AST, ist nur begrenzt dokumentiert und liefert darüber hinaus keinen eigenen Lexer mit.

ANTLR erzeugt, im Gegensatz zu den letzten drei genannten Projekten, auch automatisch einen AST und entspricht lediglich bei der Abfrage der Position des Token nicht ganz unseren Vorstellungen.

An dieser Stelle sei ein Ausblick auf den (soweit möglich) abschätzbaren Aufwand der nächsten Schritte geworfen. Bei allen Tools wäre es notwendig, eine Funktionalität zu implementieren, welche die Kommentare mit in den AST aufnimmt.

Bei JavaCC stünde zusätzlich eine Funktionalität zur Erzeugung des AST sowie eine Korrektur der Grammatik aus.

Bei CUP wäre ebenfalls eine Implementierung der AST-Erzeugung und die Erstellung eines bracuhbaren Lexers notwendig.

Für Coco/R müsste eine Grammatik für Java 1.5 erzeugt werden und ebenfalls die Funktionalität der AST-Erzeugung implementiert werden.

Bei ANTLR wäre es lediglich notwendig, die Funktion zur Positionsabfrage zu modifizieren.

Aufgrund der genannten Fähigkeiten und des geschätzten Aufwands im Vergleich der einzelnen Tools ist ANTLR unser derzeitiger Favorit.

C.7 Javaquelltexte für Tests

```
1  /**
2   * Diese Klasse wurde von Eric Blake geschrieben und enthaelt
3   * einige Graenzfalle der Javaspezifikation, bis Version 1.4.
4   * Stammt urspruenglich aus Testquelltexten fuer die Java 1.5
5   * Grammatik von CUP.
6   */
7
8  /** Some valid java code from Eric Blake.  Some of these
9   *  constructions broke previous versions of the grammars.
10  *  These should all compile with any JLS2 javac, as well as
11  *  parse correctly (no syntax errors) using the
12  * java12.cup/java14.cup/java15.cup grammars in this package.*/
13 class Eric{
14 // parenthesized variables on the left-hand-size of
15 // assignments are legal according to JLS 2.  See comments
```

```
16 // on jikes bug 105
17 // www−124.ibm.com/developerworks/bugs/?func=detailbug&bug
18 // _id=105&group_id=10
19 // for more details. According to Eric Blake:
20 //  The 2nd edition JLS is weak on this point − the grammar
21 //  in 15.26 prohibits assignments to parenthesized
22 //  variables, but earlier in 15.8.5 it states that a
23 //  parenthesized variable is still a variable (in JLS1, a
24 //  parenthesized variable was a value), and the intent of
25 //  assignment is that a variable appear on the left hand
26 //  side.  Also, the grammar in chapter 18 (if you can call
27 //  it such, because of its numerous typos and ambiguities)
28 //  permits assignment to parenthesized variables.
29     void m(int i){
30        (i) = 1;
31     }
32 // array access of an initialized array creation is legal;
33 // see Sun bugs 4091602, 4321177:
34 // developer.java.sun.com/developer/bugParade/bugs/4091602.html
35 // developer.java.sun.com/developer/bugParade/bugs/4321177.html
36 // Eric Blake says:
37 //  Again, the body of the JLS prohibits this, but chapter 18
38 //  permits it.
39     int i = new int[]{0}[0];
40     int j = new char[] { 'O', 'K' }.length;
41
42 // plain identifiers can qualify instance creation and
43 // explicit constructors; see Sun bug 4750181:
44 // developer.java.sun.com/developer/bugParade/bugs/4750181.html
45 // Eric Blake says:
46 //  Sun admits the grammars between the earlier chapters and
47 //  chapter 18 are incompatible, so they are not sure whether
48 //  things like "identifier.new name()" should be legal or
49 //  not. Chapter 18 treats identifiers as primaries, and javac
50 //  compiles them.
51     class B { };
52     B b;
53     void foo(Eric e) {
54        e.b = e.new B();
55     }
56 }
```

Listing 4: Eric.java

```
1 /**
```

```
2  * Diese Klasse enthaelt Escapesequenzen.
3  * Stammt urspruenglich aus Testquelltexten fuer die Java
4  * 1.5 Grammatik von CUP.
5  */
6 class EscapeSequences
7 {
8    char c = '\u0009';
9 }
```

Listing 5: EscapeSequences.java

```
1  /**
2   *  Diese Klasse enthaelt zulaessige Generics−Konstrukte.
3   */
4  public class AllowedGenericConstruct<T>
5  {
6     class B<S>
7     {
8        //empty
9     }
10    AllowedGenericConstruct<Integer>.B<Integer> c;
11 }
```

Listing 6: AllowedGenericConstruct.java

```
1  import java.io.*;
2  /**
3   * Diese Klasse benutzt ein ForEach−Konstrukt, welches in der
4   * endgueltigen Fassung von Java 5 erlaubt ist.
5   */
6  public abstract class AllowedForEachConstructs{
7
8     public void printArray()
9     {
10       int[][] iaa = new int[10][10];
11       for (int ia[] : iaa)
12       {
13          for (int i : ia)
14          {
15             System.out.print(i);
16          }
17       }
18    }
19 }
```

Listing 7: AllowedForEachConstructs.java

```
1  import java.io.*;
2  /**
3   * Diese Klasse benutzt ein ForEach Konstrukt, welches in der
4   * endgueltigen Fassung von Java 5 erlaubt ist. seihe:
5   * http://java.sun.com/developer/technicalArticles/releases/
6   * j2se15langfeat/
7   */
8  public abstract class AnotherAllowedForEachConstruct{
9
10    public void newFor(Collection<String> c)
11    {
12       for(String str : c)
13       {
14          sb.append(str);
15       }
16    }
17 }
```

Listing 8: AnotherAllowedForEachConstruct.java

```
1  /** A valid JSR−14 Java program, which illustrates some
2   * corner−cases in the 'smart lexer' lookahead implementation
3   * of the grammar. It should compile correctly using a
4   * JSR−14 javac, as well as parse correctly (no syntax
5   * errors) using the java15.cup grammar in this package.
6   */
7  public class Test15<X> {
8     <T> Test15(T t) { }
9     int a = 1, b = 2;
10    C c1 = new C<Integer>(),
11    c2 = new C<B>(),
12    c3 = new C<B[]>();
13    C<B> cc2 = c2;
14    C<B[]> cc3 = c3;
15    boolean d = a < b, e = a < b;
16    int f[] = new int[5];
17    boolean g = a < f[1];
18    boolean h = ( a < f[1] );
19    Object i0 = (A) cc3;
20    Object  i = ( A < B[] > ) cc3;
21    Object  j = ( A < B > ) cc2;
```

```
22  Object  k = ( A < A < B[] > >) null;
23  Object  kk= ( A < A < B[] >>) null;
24  Test15<X>.H hh = null;
25  {
26     Test15<X>.H hhh = null;
27     for (boolean l=a<b, m=a<b; a<b ; l=a<b, f[0]++)
28        a=a;
29     for (;d;)
30        b=b;
31     A<Integer> m = c1;
32     if (m instanceof C<Integer>)
33        a=a;
34     for (boolean n = m instanceof C<Integer>,
35           o = a<b,
36           p = cc3 instanceof C<B[]>;
37         cc3 instanceof C<B[]>;
38         n = m instanceof C<Integer>,
39           o = a<b,
40           p = cc3 instanceof C<B[]>)
41        b=b;
42     for (;m instanceof C<Integer>;)
43        a=a;
44     if (a < b >> 1)
45        ;
46     Object o1 = new A<A<B>>(),
47        o2 = new A<A<A<B>>>(),
48        o3 = new A<A<D<B,A<B>>>>();
49
50     // new, "explicit parameter" version of method
51  // invocation.
52     A<Integer> aa = Test15.<A<Integer>>foo();
53     /* although the spec says this should work:
54       A<Integer> aa_ = <A<Integer>>foo();
55      * Neal Gafter has assured me that this is a bug in the
56   * spec.
57      * Type arguments are only valid after a dot. */
58
59     // "explicit parameters" with constructor invocations.
60  // prototype 2.2 chokes on this.
61     new <String> K<Integer>("xh");
62     this.new <String> K<Integer>("xh");
63  }
64
65  static class A<T> { T t; }
```

```
66    static class B { }
67    static class C<T> extends A<T> { }
68    static class D<A,B> { }
69    static class E<X,Y extends A<X>> { }
70    static interface F { }
71    // wildcard bounds.
72    static class G{
73       A<? extends F> a; A<? super C<Integer>> b;
74    }
75    class H { }
76    static class I extends A<Object[]> { }
77    static class J extends A<byte[]> { }
78    class K<Y> { <T>K(T t) { Test15.<T>foo(); } }
79
80    static <T> T foo() { return null; }
81 }
```

Listing 9: Test15.java

```
1  /**
2   * Diese Klasse ist ein Testquelltext fuer die Java 1.5
3   * Grammatik von CUP. Das Original wird von CUP fehlerfrei
4   * geparst, ist jedoch nicht mit javac kompilierbar.
5   * Nachdem wir einige Auskommentierungen vorgenommen
6   * hatten, wurde diese auch kommentarlos von javac
7   * kompiliert. Da diese Klasse recht komplexe Statements
8   * enthaelt, wird diese auch fuer die Parsingtests benutzt.
9   * Dies ist jedoch mehr ein statistischer Test.
10  */
11 import static java.lang.Math.*; // test of static import
12 import static java.lang.System.out; // ditto
13 import java.util.*;
14
15 class TestJSR201Berichtigt
16 {
17    enum Color { red, green, blue ; };
18
19    public static void main(String... args/* varargs */)
20    {
21       /* for each on multi-dimensional array */
22       int[][] iaa = new int[10][10];
23       for (int ia[] : iaa)
24       {
25          for (int i : ia)
26             out.print(i); // use static import.
```

```
27            out.println();
28        }
29        /* */
30 /////////////////////////////////////////////////////////
31 // Auskommentiert da der Compiler Color.VALUES nicht kennt
32 /////////////////////////////////////////////////////////
33 //       for (Color c : Color.VALUES)
34 //       {
35 //           switch(c)
36 //           {
37 //               case Color.red: out.print("R");
38 //               break;
39 //               case Color.green: out.print("G");
40 //               break;
41 //               case Color.blue: out.print("B");
42 //               break;
43 //               default: assert false;
44 //           }
45 //       }
46 /////////////////////////////////////////////////////////
47        out.println();
48    }// end of main
49
50    // complex enum declaration, from JSR-201
51    public static enum Coin
52    {
53        penny(1), nickel(5), dime(10), quarter(25);
54        Coin(int value)
55        {
56            this.value = value;
57        }
58        private final int value;
59        public int value()
60        {
61            return value;
62        }
63    }
64
65    public static class Card implements Comparable,
66                            java.io.Serializable
67    {
68        public enum Rank
69        {
70            deuce, three, four, five, six, seven,
```

```
71          eight, nine, ten, jack, queen, king, ace
72      }
73
74      public enum Suit
75      {
76          clubs, diamonds, hearts, spades
77      }
78
79      private final Rank rank;
80      private final Suit suit;
81
82      private Card(Rank rank, Suit suit)
83      {
84          if (rank == null || suit == null)
85              throw new NullPointerException(rank + ",␣" +
86                                              suit);
87          this.rank = rank;
88          this.suit = suit;
89      }
90
91      public Rank rank()
92      {
93          return rank;
94      }
95
96      public Suit suit()
97      {
98          return suit;
99      }
100
101     public String toString()
102     {
103         return rank + "␣of␣" + suit;
104     }
105
106     public int compareTo(Object o)
107     {
108         Card c = (Card)o;
109         int rankCompare = rank.compareTo(c.rank);
110         return rankCompare != 0 ? rankCompare :
111         suit.compareTo(c.suit);
112     }
113
114     private static List sortedDeck = new ArrayList(52);
```

```
115
116         /* BROKEN IN PROTOTYPE 2.0 */
117 ///////////////////////////////////////////////////////////
118 // Auskommentiert da der Compiler Rank.VALUES und
119 // Suit.VALUES nicht kennt
120 ///////////////////////////////////////////////////////////
121 //        static
122 //        {
123 //            for (Rank rank : Rank.VALUES)
124 //            for (Suit suit : Suit.VALUES)
125 //                sortedDeck.add(new Card(rank, suit));
126 //        }
127 ///////////////////////////////////////////////////////////
128         /* */
129
130         // Returns a shuffled deck
131         public static List newDeck()
132         {
133             List result = new ArrayList(sortedDeck);
134             Collections.shuffle(result);
135             return result;
136         }
137     } // end of class Card
138
139     // sophisticated example:
140 ///////////////////////////////////////////////////////////
141 // abstract auskommentiert da fuer enum nicht zulaessig
142 ///////////////////////////////////////////////////////////
143     public static /*abstract*/ enum Operation
144     {
145         plus
146         {
147             double eval(double x, double y)
148             {
149                 return x + y;
150             }
151         },
152         minus
153         {
154             double eval(double x, double y)
155             {
156                 return x - y;
157             }
158         },
```

```
159        times
160        {
161           double eval(double x, double y)
162           {
163              return x * y;
164           }
165        },
166        divided_by
167        {
168           double eval(double x, double y)
169           {
170              return x / y;
171           }
172        };
173
174 // Perform arithmetic operation represented by this constant
175        abstract double eval(double x, double y);
176
177        public static void main(String args[])
178        {
179           double x = Double.parseDouble(args[0]);
180           double y = Double.parseDouble(args[1]);
181 ////////////////////////////////////////////////////////////
182 // Auskommentiert da in Compiler VALUES nicht kennt
183 ////////////////////////////////////////////////////////////
184 //          for (Operation op : VALUES)
185 //             out.println(x + " " + op + " " +
186 //                         y + " = " + op.eval(x, y));
187 ////////////////////////////////////////////////////////////
188        }
189     }
190 }// end of class TestJSR201Berichtigt
```

Listing 10: TestJSR201Berichtigt.java

```
1  /**
2   * Diese Klasse definiert ein abstraktes enum, welches nicht
3   * erlaubt ist. Erzeugt beim Kompilieren mit javac
4   * folgenden Fehler:
5   *
6   * AbstractEnumShouldGiveParseError.java:3:
7   *    modifier abstract not allowed here
8   *    public static abstract enum Operation
9   *                        ^
10  * 1 error
```

```
11  *
12  */
13  class AbstractEnumShouldGiveParseError{
14
15     public static abstract enum Operation
16     {
17        plus
18        {
19           double eval(double x, double y)
20           {
21              return x + y;
22           }
23        },
24        minus
25        {
26           double eval(double x, double y)
27           {
28              return x − y;
29           }
30        },
31        times
32        {
33           double eval(double x, double y)
34           {
35              return x * y;
36           }
37        },
38        divided_by
39        {
40           double eval(double x, double y)
41           {
42              return x / y;
43           }
44        };
45
46  // Perform arithmetic operation represented by this constant
47        abstract double eval(double x, double y);
48
49        public static void main(String args[])
50        {
51           double x = Double.parseDouble(args[0]);
52           double y = Double.parseDouble(args[1]);
53        }
54     }
```

```
55 }
```

Listing 11: AbstractEnumShouldGiveParseError.java

```
 1 /**
 2  * Diese Klasse kombiniert die Modifier abstract und final,
 3  * welche sich gegenseitig ausschliessen.
 4  * Erzeugt beim Kompilieren mit javac folgenden Fehler:
 5  *
 6  * AbstractFinalShouldGiveParseError.java:20:
 7  * illegal combination of modifiers: abstract and final
 8  *     public abstract final void someAbstractMethod();
 9  *                                ^
10  * 1 error
11  *
12  */
13 import java.io.*;
14
15 public abstract class AbstractFinalShouldGiveParseError{
16
17    public abstract final void someAbstractMethod();
18 }
```

Listing 12: AbstractFinalShouldGiveParseError.java

```
 1 /**
 2  * Diese Klasse nutzt ein for each ( in ) Konstrukt, welches
 3  * in den Previews zu Java 5 Tiger noch unterstuetzt wurde, es
 4  * allerdings nicht in die endgueltige Fassung geschafft hat.
 5  * Erzeugt beim Kompilieren mit javac folgenden Fehler:
 6  *
 7  * ForEachInShouldGiveParseError.java:13: '(' expected
 8  *         for each (int ia[] in iaa)
 9  *             ^
10  * ForEachInShouldGiveParseError.java:20:
11  *  illegal start of expression
12  *     }
13  *     ^
14  * 2 errors
15  *
16  */
17 import java.io.*;
18
19 public class ForEachInShouldGiveParseError{
20
```

```
21      public void printArray()
22      {
23          int[][] iaa = new int[10][10];
24          for each (int ia[] in iaa)
25          {
26              for each (int i in ia)
27              {
28                  System.out.println(i);
29              }
30          }
31      }
32  }
```

Listing 13: ForEachInShouldGiveParseError.java

```
1   /**
2    * Diese Klasse kombiniert die Modifier public und private,
3    * welche sich gegenseitig ausschliessen. Erzeugt beim
4    * Kompilieren mit javac folgenden Fehler:
5    *
6    * PublicPrivateShouldGiveParseError.java:15:
7    *  illegal combination of modifiers: public and private
8    *     public private void someMethod();
9    *                        ^
10   * 1 error
11   */
12  import java.io.*;
13
14  public class PublicPrivateShouldGiveParseError{
15
16      public private void someMethod()
17      {
18          // do nothing
19      }
20  }
```

Listing 14: PublicPrivateShouldGiveParseError.java

D Erweiterte Tests des ANTLR-generierten Parsers

D.1 Beschreibung

Um festzustellen, ob der mit ANTLR erzeugte Parser bis hierhin noch nicht aufgedeckte Schwachstellen oder Fehler besitzt, war es nun notwendig, weitere Tests durchzuführen. Ebenso war es notwendig, den Parser auf seine Verarbeitungsgeschwindigkeit hin zu prüfen.

Alle im folgenden genannten Ergebnisse beziehen sich auf den Parser, welcher aus der ANTLR-Grammatik für Java 1.5 von Michael Studman erzeugt wurden.Die beiden anderen Grammatiken (Stahl, Wisniewski) sind aufgrund der bereits im Test der verschiedenen Tools gefundenen Fehler ausgeschieden.

Um die Analyse dieser erweiterten Tests zu erleichtern, war es zunächst nötig, das verwendete Testprogramm zu modifizieren. Da die Konsole, sowohl unter Windows als auch unter Linux, aufgrund der begrenzten maximalen Zahl der Ausgaben und der beschränkten Breite nicht gerade ideal bei der Analyse größerer Codemengen ist, wurden jegliche Ausgaben in eine Textdatei geschrieben anstatt auf die Konsole. Desweiteren wurde die Verarbeitungszeit gemessen und ebenfalls in der Ausgabedatei gespeichert. Der zugehörige Code findet sich in Listing 16.

D.2 Breitentests

Da es unmöglich ist, alle in Java möglichen Kombinationen von Sprachkonstrukten mit einem vertretbaren Aufwand in einzelnen Testfällen durchzuspielen, und wir auch nichts in der Art einer „Testsuite“ für Java-Parser oder -Compiler finden konnten, haben wir einige größere, quelloffen verfügbare, Projekte parsen lassen.

D.2.1 JDK

Die Quelltexte des Java Developer Kits (Version 1.5.0_06) waren die Wahl für den ersten Test. Die insgesamt 6.555 Dateien ergeben ein Datenvolumen von knapp 63 MB. Das Parsing verlief vollständig fehlerfrei, alle Dateien wurden ohne Probleme akzeptiert.

D.2.2 Eclipse

Zum zweiten Test haben wir die Quelltexte des Eclipse SDK (Version 3.1.2) verwendet, welche knapp 90 MB groß sind und sich auf 12.304 Dateien verteilen. Der Test verlief hier fast völlig fehlerfrei, lediglich 4 Dateien konnten nicht geparst werden. Allerdings konnten eben diese 4 in der vorliegenden Form auch nicht direkt mit javac kompiliert werden. Der Grund

hierfür liegt in den enthaltenen Quelltexten, welche Code enthälten, der Konstrukte wie in Listing 15 enthält.

```
%Options in the template
%hasDefault

%if hasDefault
   [...]
%else
   [...]
%endif
```

Listing 15: Ein Beispiel für ant-spezifischen Code in den Eclipse-SDK-Quelltexten

Dies ist allerdings kein gültiger Javacode. Da die Erzeugung von Eclipse mittels ant (einer Art make-Tool) geschieht, wird dieser Code jedoch vor der eigentlich Kompilierung nochmals modifiziert. Somit sind die vier aufgetretenen Fehler kein Indiz für ein Versagen oder eine Unvollständigkeit des Parsers, sondern durchaus berechtigt.

D.2.3 ANTLR

Im Vergleich mit dem JDK und Eclipse sind die Quelltexte von ANLR mit 307 Dateien und etwa 2 MB eher klein. Das Parsing verläuft hier bis auf eine Datei fehlerfrei. Diese Datei befindet sich jedoch lediglich unter den mitgelieferten Beispielen von ANTLR und kann auch mit javac nicht kompiliert werden, so dass auch hier der Grund für das Versagen schlicht und einfach falscher Code ist und nicht der Parser.

D.2.4 Fazit

Beim Parsen von existierenden Anwendungen konnten alle Quelltexte fehlerfrei geparst werden, die auch in der vorliegenden Form mit dem Java Compiler erzeugt werden konnten, so dass wir diese Tests als vollständig bestanden bewerten.

D.3 Test mit Textinternationalisierungsoptionen

Da es bei den ersten Tests aller Tools bereits auffällig oft an den Quelltexten mit Unicode bzw. nicht-ASCII-Text zu Fehlern kam, haben wir auch hier weitergehende Tests durchgeführt.

D.3.1 Unicode-Bezeichner

Beim ersten Test mit Unicode hatten wir lediglich auf die Fälle getestet, in denen Unicode bzw. Unicode-Escapesequenzen in den zugewiesenen Werten von `String`- und `char`-Variablen vorhanden waren. Da Java 5 solche Zeichen aber auch an anderen Stellen, wie Bezeichnern von Variablen und Klassen, zulässt, haben wir weitere Tests durchgeführt. Dabei haben wir Klassen- und Variablenbezeichner sowohl mit Sonderzeichen (siehe Listing 17) als auch den gleichen Code nochmals mit den entsprechenden Unicode-Escapesequenzen (siehe Listing 18) durchgeführt.

Beide Dateien ließen sich mit javac kompilieren, der Parser scheitert jedoch in beiden Fällen beim ersten Bezeichner. Die verwendeten Quelltexte befinden sich auch hier im Abschnitt D.5.

D.3.2 Textencoding

Bei diesen Tests haben wir einen einfachen Quellcode, welcher sich fehlerfrei parsen ließ, in verschiedenen Textformaten abgespeichert. Dabei kamen neben dem normalen ASCII auch die verschiedenen Unicode-Varianten (Unicode, Big Endian, UTF-8, UTF-7) und ANSI-Varianten mit verschiedenen Codepages (31 an der Zahl) zum Einsatz.

Zunächst haben wir die Formate aussortiert, die auch javac nicht verarbeiten konnte. Dabei fielen alle Unicode-Varianten und die ASCII-Varianten mit den Codepages 37, 500, 875 und 1026 heraus. Alle übrig gebliebenen akzeptierte der Parser anstandslos. Ein testweises Parsen der 8 von javac nicht akzeptierten Formate schlug in allen Fällen ebenfalls fehl. Tabelle 24 zeigt eine Übersicht über alle getesteten Formate.

Encoding	**Codepage**	**Beschreibung**	**ANTLR**	**javac**
ANSI	37	IBM EBCDIC (US-Kanada)	nein	nein
ANSI	437	OEM USA	ja	ja
ANSI	500	IBM EBCDIC (International)	nein	nein
ANSI	737	Griechisch (DOS)	ja	ja
ANSI	775	Baltisch (DOS)	ja	ja
ANSI	850	Westeuropäisch (DOS)	ja	ja
ANSI	852	Osteuropäisch (DOS)	ja	ja
ANSI	855	OEM kyrillisch	ja	ja
ANSI	857	Türkisch (DOS)	ja	ja

ANSI	860	Portugiesisch (DOS)	ja	ja
ANSI	861	Isländisch (DOS)	ja	ja
ANSI	863	Französisch, Kanada (DOS)	ja	ja
ANSI	865	Nordisch (DOS)	ja	ja
ANSI	866	Kyrillisch (DOS)	ja	ja
ANSI	869	Griechisch, modern (DOS)	ja	ja
ANSI	874	Thai (Windows)	ja	ja
ANSI	875	IBM EBCDIC (Griechisch, modern)	nein	nein
ANSI	932	Japanisch (Shift-JIS)	ja	ja
ANSI	936	Chinesisch - VR (GB2312)	ja	ja
ANSI	949	Koreanisch	ja	ja
ANSI	950	Chinesisch (Traditionell) (Big5)	ja	ja
ANSI	1026	IBM EBCDIC (Türkisch, Latin-5)	nein	nein
ANSI	1250	Mitteleuropäisch (Windows)	ja	ja
ANSI	1251	Kyrillisch (Windows)	ja	ja
ANSI	1252	Westeuropäisch (Windows)	ja	ja
ANSI	1253	Griechisch (Windows)	ja	ja
ANSI	1254	Türkisch (Windows)	ja	ja
ANSI	1255	Hebräisch (Windows)	ja	ja
ANSI	1256	Arabisch (Windows)	ja	ja
ANSI	1257	Baltisch (Windows)	ja	ja
ANSI	1258	Vietnamesisch (Windows)	ja	ja
US-ASCII	-	-	ja	ja
Unicode (UTF-7)	-	-	nein	nein
Unicode (UTF-8)	-	-	nein	nein
Unicode	-	-	nein	nein
Unicode (Big Endian)	-	-	nein	nein

Tabelle 24: Übersicht aller getesteten Textencodings

D.3.3 Fazit

Bei den möglichen Encoding-Verfahren für Quelltext akzeptiert der Parser alles, was auch vom Java Compiler verarbeitet werden kann. Allerdings schlägt das Parsing bei der Verwendung von Sonderzeichen und Unicode-Escapesequenzen in Klassen-, Variablen-, Methodenbezeichnern etc. komplett fehl, so dass die Grammatik in dieser Hinsicht eventuell noch überarbeitet werden müsste.

D.4 Benchmarks

An dieser Stelle haben wir die Verarbeitungsgeschwindigkeit des Parsers untersucht, indem wir die Zeiten für das Parsing der kompletten Quelltexte des JDK bzw. von Eclipse gemessen haben (nähere Informationen über deren Umfang finden sich weiter oben im Abschnitt der Breitentests).

Mit einer Verarbeitungsdauer von 1:06 Minuten für das JDK und 1:58 Minuten für Eclipse (auf einem System mit 3,2GHz Intel P4) kann man durchaus von einer sehr schnellen Verarbeitung sprechen, so dass ein von ANTLR erzeugter Parser als ausreichend schnelle Grundlage für den Faktenextraktor betrachtet werden kann.

D.5 Quelltexte

```
1  import java.io.*;
2  import antlr.collections.AST;
3  import antlr.collections.impl.*;
4  import antlr.debug.misc.*;
5  import antlr.*;
6  import java.awt.event.*;
7  import java.util.*;
8
9  class LogParser {
10
11  static boolean showTree = false;
12
13  static StringBuilder resultOutPut = new StringBuilder();
14
15    public static void main(String[] args) {
16   // Use a try/catch block for parser exceptions
17   try {
18    // if we have at least one command-line argument
19    if (args.length > 0 ) {
20     // get start time
21     Date startdate = new Date();
22     outputappend("Startzeit:␣" + startdate.toString());
23     System.err.println("Parsing...");
24
25     // for each directory/file specified on the command line
26     for(int i=0; i< args.length;i++) {
27      if ( args[i].equals("-showtree") ) {
28       showTree = true;
29      }
```

```
30      else {
31       outputappend("␣" + args[i] + ":");
32       doFile(new File(args[i]) , args[i]); // parse it
33      }
34     }
35     // get end time & calculate difference
36     Date enddate = new Date();
37     outputappend("Endzeit:␣" + enddate.toString());
38     long timerequired = enddate.getTime() − startdate.getTime();
39     outputappend("Zeit␣benötigt:␣" +
40       Long.toString(timerequired / 3600000) + ":" +
41       Long.toString((timerequired \% 3600000) / 60000) + ":" +
42       Long.toString((timerequired \% 60000) / 1000) + "," +
43       Long.toString(timerequired \% 1000) +
44       "␣(=" + Long.toString(timerequired) + "ms)");
45     FileOutputStream mystream =
46       new FileOutputStream("ParserLog.txt");
47     mystream.write (resultOutPut.toString().getBytes() , 0 ,
48       resultOutPut.length());
49     mystream.close();
50    }
51    else
52     System.err.println("Usage:␣java␣Main␣[−showtree]␣"+
53                              "<directory␣or␣file␣name>");
54   }
55   catch(Exception e) {
56    System.err.println("exception:␣"+e);
57    e.printStackTrace(System.err);   // so we can get stack trace
58   }
59  }
60
61
62  private static void outputappend (String toAppend) {
63   resultOutPut.append(toAppend + "\n");
64  }
65
66
67  private static String stackTraceToString(Exception e) {
68   String myresult = "";
69   StackTraceElement trace[] = e.getStackTrace();
70   for (int i = 0; i < trace.length; i++) {
71    myresult += "@␣" + trace[i].toString() + "\n";
72   }
73   return myresult;
```

```
74  }
75
76
77  // This method decides what action to take based on
78  //   the type of file we are looking at
79  public static void doFile(File f , String basepath)
80          throws Exception {
81   // If this is a directory, walk each file/dir within
82   if (f.isDirectory()) {
83    String files[] = f.list();
84    for(int i=0; i < files.length; i++)
85     doFile(new File(f, files[i]) , basepath);
86   }
87
88   // otherwise, if this is a java file, parse it!
89   else if ((f.getName().length()>5) &&
90      f.getName().substring(f.getName().length()−5).equals(".java")) {
91    outputappend("␣␣␣␣." +
92      f.getAbsolutePath().substring(basepath.length()) );
93    System.err.print(".");
94    // parseFile(f.getName(), new FileInputStream(f));
95    parseFile(f.getName(),
96      new BufferedReader(new FileReader(f)));
97   }
98  }
99
100 // Here's where we do the real work...
101 public static void parseFile(String f, Reader r)
102         throws Exception {
103  try {
104   // Create a scanner that reads from the
105   //   input stream passed to us
106   JavaLexer lexer = new JavaLexer(r);
107   lexer.setFilename(f);
108
109   // Create a parser that reads from the scanner
110   JavaRecognizer parser = new JavaRecognizer(lexer);
111   parser.setFilename(f);
112
113   // start parsing at the compilationUnit rule
114   parser.compilationUnit();
115
116   // do something with the tree
117   doTreeAction(f, parser.getAST(), parser.getTokenNames());
```

```
118   }
119   catch (Exception e) {
120    outputappend("parser␣exception:␣"+e);
121    outputappend("message:␣" + e.toString());
122    outputappend(stackTraceToString(e));
123    System.err.print("!");
124    // e.printStackTrace();   // so we can get stack trace
125   }
126  }
127
128  public static void doTreeAction(String f, AST t,
129    String[] tokenNames) {
130   if ( t==null ) return;
131   if ( showTree ) {
132    ((CommonAST)t).setVerboseStringConversion(true, tokenNames);
133    ASTFactory factory = new ASTFactory();
134    AST r = factory.create(0,"AST␣ROOT");
135    r.setFirstChild(t);
136    final ASTFrame frame = new ASTFrame("Java␣AST", r);
137    frame.setVisible(true);
138    frame.addWindowListener(
139     new WindowAdapter() {
140       public void windowClosing (WindowEvent e) {
141         frame.setVisible(false); // hide the Frame
142         frame.dispose();
143         System.exit(0);
144       }
145     }
146    );
147    // System.out.println(t.toStringList());
148   }
149   JavaTreeParser tparse = new JavaTreeParser();
150   try {
151    tparse.compilationUnit(t);
152   }
153   catch (RecognitionException e) {
154    outputappend(e.getMessage());
155    outputappend(stackTraceToString(e));
156    System.err.print("!");
157    // e.printStackTrace();%
158   }
159
160  }
161 }
```

Listing 16: LogParser.java

```
1  /**
2   * Diese Klasse enthält Unicode−Sonderzeichen.
3   */
4  class EscapeSäquänz   // "EscapeSäquänz" ;−)
5  {
6     char c = '\u0009';   // this literal is valid.
7     // "Änderungsmaßstab" ;−)
8     String Änderungsmaßstab = "gar␣keiner";
9     int français = 69;   // "français" ;−)
10 }
```

Listing 17: TestUnicodeClassAndVariableNames.java

```
1  /**
2   * Diese Klasse enthält Unicode−Escapesequenzen.
3   */
4  class EscapeS\u00e4qu\u00e4nz   // "EscapeSäquänz" ;−)
5  {
6     char c = '\u0009';   // this literal is valid.
7     // "Änderungsmaßstab" ;−)
8     String \u00c4nderungsma\u00dfstab = "gar␣keiner";
9     int fran\u00e7ais = 69;   // "français" ;−)
10 }
```

Listing 18: TestUnicodeEscapeSequenceClassAndVariableNames.java

E Detaillierte Funktionsweise von ANTLR

E.1 Grundlagen

Dieser Abschnitt beschreibt die grundlegende Funktionsweise von ANTLR, den strukturellen Aufbau der Grammatiken und die Elemente der ANTLR-Metasprache , sowie die Umsetzung dieser in Javaquelltext.

E.1.1 Funktionsweise von ANTLR

ANTLR ist ein Parsergeneratoren - es erzeugt aus einer Grammatik einen Parser. Im Gegensatz zu den meisten anderen Generatoren kann ANTLR jedoch ohne Umwege auch (zum Parser passende) Lexer und Treeparser erzeugen. Es ist dazu nicht nötig, weitere Zusatzprogramme zu Hilfe zu nehmen, wie es beispielsweise bei CUP der Fall ist[23].

Grundsätztlich gilt, dass für Lexer, Parser und Treeparser jeweils eine Grammatik geschrieben werden muss. Dabei kann der Benutzer frei wählen, welche er realisieren will. Braucht er beispielweise nur einen Lexer, so muss er nur die Lexergrammatik schreiben. Da ANTLR dem Paradigma der Objektorientierung folgt, ist es auch möglich die Grammatiken weiter zu vererben.

Zur Erzeugung der gewünschten Programme werden die Grammatiken dann per Kommandozeilenparameter an ANTLR übergeben. Den Zusammenhang der Grammatikdateien und die Funktionsweise der erzeugten Programme stellt Abbildung 33 grob dar. Der Aufbau dieser Grammatiken wird im nächsten Abschnitt konkretisiert.

E.1.2 Aufbau der Grammatiken

Alle drei Grammatikarten (für Lexer, Parser und Treeparser) sind auf die selbe Weise aufgebaut und werden in der ANTLR-Metasprache geschrieben. Ziel ist es, möglichst lesbare und somit verständliche Grammatiken zu erhalten. Dabei können die verschiedenen Grammatiken wahlweise in einer gemeinsamen Datei, jeweils in einer eigenen oder in einer Kombination davon stehen.

Die Grammatiken beginnen mit einer optionalen Präambel, es folgt verpflichtend die Definition der Parserklasse, optional Header, Optionen, Tokendefinitionen und eigene Methoden. Daran schließt sich, wieder verpflichtend, die Definition der Regeln an.

[23]CUP kann keine „eigenen“ Lexer erzeugen. Es muss zunächst ein passender Lexer geschrieben oder mit einem Scannergenerator wie JLex erzeugt werden.

Lexergrammatik

```
class MyLexer extends Lexer;

options{
   lookahead = 2;
}

WS : TAB | BREAK;
ID : NAME | TITLE;
```

Parsergrammatik

```
class MyParser extends Parser;

options{
   buildAST = true;
}

stmt : type ID^ SEMI!;
type : INT | DOUBLE | FOO;
```

Treeparsergrammatik

```
class MyTreeparser extends Treeparser;

options{
   lookahead = 2;
}

line : #(ID t:type) {t.getText();}
```

ANTLR

Input

MyLexer

Tokenstream

MyParser

AST

MyTreeparser

Output

Abbildung 33: Funktionsweise von ANTLR

Präambel
Definition der Parserklasse
Header
Optionen
Tokendefinitionen
Eigene Methoden
Regeldefinitionen

Abbildung 34: Aufbau einer Grammatik

Präambel. Der Inhalt aller Zeilen vor der Definition der Parserklasse nennt sich die Präambel. Diese wird unverändert vor den Quelltext der erzeugten Klasse kopiert (also noch vor die Definition der Klasse). Dies dient z. B. dem Einfügen von Kommentaren, Importklauseln für zusätzlich benötigte Klassen oder einer Packagedeklaration in Java.

Definition der Parserklasse. Diese ähnelt dem Aussehen einer Klassendefinition (mit Ableitung einer Oberlasse) in Java und legt fest, ob es sich dabei um eine Grammatik für einen Lexer, Parser oder Treeparser handelt (Beispiel siehe Listing 19). Tatsächlich führt standardmässig, in Java, die Angabe von `Lexer` zur Ableitung der Klasse `antlr.CharScanner`, `Parser` zur Ableitung von `antlr.LLkParser` und `TreeParser` zur Ableitung von `antlr.TreeParser`. Wird eine andere Klasse benötigt, so kann diese, in Klammern und Anführungszeichen nach der Definition, angegeben werden (Beispiel siehe Listing 20). Diese muss selbst allerdings eine Subklasse der jeweiligen oben angeführten Klassen sein.

```
class MyLexer extends Lexer;
```

Listing 19: Definition und somit Beginn einer Lexergrammatik

```
class OtherLexer extends Lexer("antlr.debug.DebuggingCharScanner")
    ;
```

Listing 20: Benutzung eines anderen Lexers

Alle nun folgenden Abschnitte, mit Ausnahme des Abschnittes der Regeln, werden in geschweiften Klammern eingefasst.

Header. Im Header kann eigener Quelltext definiert werden, der in die zu erzeugenden Klassen aufgenommen werden soll (unmittelbar nach der Klassendefinition). Dieser spielt meist nur für C++ basierte Parser eine Rolle, da in dieser Sprache Elemente erst deklariert werden müssen, bevor diese referenziert werden dürfen (wie in Listing 21).

```
{
  String * message;  // ein Zeiger auf einen String
}
```

Listing 21: Mögliche Verwendung für einen Header

Optionen. In den Optionen werden abschnittsspezifische Einstellungen vorgenommen. Abschnittsspezifisch, da Optionen von Datei-, über Grammatik-, bis zu Regel- und Tokenebene hinab, gesetzt werden können. In den von uns benutzten Grammatiken wurden Optionen allerdings nur auf Grammatik- und Regelebene gesetzt. Ein kurzes Beispiel ist in den Listings 22 und 23 zu sehen. Eine komplette Aufstellung aller Optionen befindet sich auf der Hompepage von ANTLR unter [16].

```
options{
  exportVocab=MyVocabulary; // Name des Tokenvokabulars
}
```

Listing 22: Optionsabschnitt einer Lexergrammatik

```
options{
  k = 2; // Größe des Lookaheads
  buildAST = true; // Parser soll AST erzeugen
  importVocab = MyVocabulary; // Zu benutzendes Vokabular an
     Tokens
}
```

Listing 23: Optionsabschnitt einer Parsergrammatik

Tokendefinitionen. Die Tokendefinitionen dienen für ggf. nötige Anpassungen der Tokens an die Grammatik. I. d. R. werden die Tokens im Lexer erzeugt und deshalb auch in dessen Grammatik definiert. Diese Tokens stehen dann zur Verwendung in weiteren Grammatiken zur Verfügung, wenn die Option `exportVocab` gesetzt wurde, wie in Listing 22 gezeigt. Mit der Option `importVocab` (siehe Listing 23) wird das Vokabular in einer anderen Grammatik nutzbar gemacht.

Für den Fall, dass beispielsweise im Vokabular benötigte Tokens fehlen, oder diese anderen Werten zugeordnet werden sollen, können die Tokens in den Tokendefinitionen angepasst werden.

Dies kann nicht nur bei der Verebung von Grammatiken vorkommen, sondern auch bei einfacher Verwendung des Vokabulars. Da im Treeparser nur AST-Elemente angesprochen werden können, die auf einem Token basieren, werden oft „imaginäre“ Tokens definiert, um beispielsweise abstraktere Strukturen ansprechen zu können.

```
tokens{
  EXPR;  // ein kompletter Ausdruck
  DECL;  // eine Deklaration
}
```

Listing 24: Tokendefinitionen

Eigene Methoden. Die eigenen Methoden müssen in der Zielsprache geschrieben sein, da diese unverändert in den Quelltext der erzeugten Klasse (nach dem Inhalt aus dem Header) kopiert werden. Diese können dann beispielsweise in den semantischen Aktionen von Grammatikregeln zum Einsatz kommen.

Regeldefinitionen. Grundsätzlich wird, wie auch die Präambel, dieser letzte Teil nicht in geschweiften Klammern eingefasst. Es handelt sich dabei i. d. R. um den umfangreichsten Abschnitt. Deshalb wird den Regeldefinitionen ein eigener Abschnitt gewidmet.

E.1.3 ANTLR-Metasprache

Alle Regeln werden in einer EBNF-ähnlichen Notation definiert. Zu den Elementen der EBNF kommen ergänzende Operatoren und neue Elemente für Sichtbarkeit, Exceptions, semantische Aktionen, Eingabe- & Ausgabeparameter sowie syntaktische & semantische Prädikate hinzu. Diese Menge von Elementen bildet die ANTLR-Metasprache (näheres siehe [17]). Das in den folgenden Paragraphen jeweils beschriebene Element ist, im dazugehörigen Beispiel, farblich hervorgehoben - Schlüsselworte sind unterstrichen.

Operatoren. Grundsätzlich sind alle syntaktischen Elemente der EBNF vertreten. Im Gegensatz zur „richtigen“ EBNF werden in der ANTLR-Metasprache eckige und geschweifte Klammern jedoch für andere Zwecke verwendet. Deshalb werden diese durch andere Konstrukte ersetzt und ergänzt. Tabelle 25 listet diese auf.

Zu den ersten vier Operatoren existieren in der EBNF unmittelbare Pendants. Der Bereichsoperator (...) ist jedoch nur durch eine verkettete Veroderung in EBNF realisierbar. Er dient der Bequemlichkeit des Grammatikschreibers - es genügt zum Beispiel `'a' .. 'z'` anzugeben, statt alle Zeichen des Alphabets einzeln aufzuführen.

In ANTLR	Semantik	In EBNF
`( ... )*`	0 bis ∞	`{ ... }`
`( ... )?`	Option: 0 oder 1	`[ ... ]`
`( ... )+`	1 bis ∞	`Symbol { Symbol }`
`~`	Element-/Set-Komplement	−
`..`	Bereichsoperator (von bis)	
`.`	Wildcard	
`^`	AST-Wurzeloperator	
`!`	AST-Ausschlussoperator	
`{ ... }?`	Semantisches Prädikat	
`{ ... }=>`	Syntaktisches Prädikat	

Tabelle 25: Operatoren der ANTLR-Metasprache

Der Wildcard-Operator (.) stammt von den regulären Ausdrücken und „matcht" jedes Zeichen.

Die AST-Operatoren dienen zum Steuern der Erzeugung eines AST und werden in Abschnitt E.1.5 detailiert erläutert. Die Prädikate werden in bereits in Paragraphen dieses Abschnitts näher behandelt.

Aufbau einer Regel. Die Regeln der Grammatik bestehen aus einer linken und rechten Seite. Die Seiten werden durch einen Doppelpunkt (:) getrennt (im Gegensatz zu „::=" in EBNF). Die Regel endet dann immer mit einem Semikolon (;).

Auf der linken Seite steht der Regelname - optional können neben der Sichtbarkeit der Regel auch Eingabe- und Ausgabeparameter, sowie eine semantische Aktionen (welche vor der Regel ausgeführt wird) festgelegt werden. Auf der rechten Seite stehen die Produktionen, welche mit Optionen, semantischen und syntaktischen Prädikaten, sowie weiteren semantischen Aktionen erweitert werden können. Ein Beispiel für eine einfache Regel zeigt Listing 25.

```
type : classOrInterfaceType | builtInType ;
```

Listing 25: Regel im Parser die eine Syntax definiert

Sichtbarkeit einer Regel. Für jede Regel kann die Sichtbarkeit per Modifizierer festgelegt werden. Da jede Regel in eine Methode umgesetzt wird, kann somit, bereits in der Grammatik, auf die spätere Sichtbarkeit Einfluss genommen werden. Es genügt dazu dem Regelnamen einen Sichtbarkeitsmodifizierer voranzustellen. Erlaubt sind `private`, `protected` und `public`.

```
private type : classOrInterfaceType | builtInType ;
```

Listing 26: Regel mit Sichtbarkeitsmodifizierer

Exceptions. Auch das Auslösen von Exceptions kann bereits in der Grammatik gesteuert werden. Durch Anhängen des Schlüsselwortes `throws`, mit der auszulösenden Exception, an den Regelnamen, wird dies festgelegt. Auf den ersten Blick macht diese Funktionalität scheinbar keinen Sinn. Zwar wird die angegebene Exception bei der Deklaration der Methode mit aufgeführt, doch wird die Exception niemals ausgelöst. Diese muss in einer semantischen Aktion manuell ausgelöst werden.

```
a throws MyException  : A ;
```

Listing 27: Eine Regel die eine Exception auslösen kann

Ferner können in einer Regel mehrere Exceptionhandler definiert werden. @TODO beschreiben/oder geht das schon zu weit?

Optionen. Wie bereits o. a. sind Optionen, bis hinab auf Regel- und Symbolebene, möglich. Um Optionen für eine Regel festzulegen, reicht, es dem Regelnamen das Schlüsselwort `options`, mit den gewünschten Optionen in geschweiften Klammern, anzuhängen. Soll eine Option nur für ein Symbol gelten, so muss vor diesem das Schlüsselwort `options`, mit den gewünschten Optionen in geschweiften Klammern und einem nachgestellten Doppelpunkt, stehen. Ferner müssen Optionsteil und Symbol in einem Klammerpaar eingefasst sein.

```
type options {defaultErrorHandler = false;} : classOrInterfaceType
                                            | builtInType ;
```

Listing 28: Option für die Regel type

```
ID : ('a'..'z')+ (options {greedy = true;}: WS)? ;
```

Listing 29: Option für das Symbol WS (Whitespace)

Semantische Aktionen. Bei einer semantischen Aktion handelt es sich um ein Quelltextfragment in der Zielsprache. Es muss in geschweiften Klammern eingefasst werden und wird (unverändert) in das zu erzeugende Programm kopiert. Somit wird es zu dem Zeitpunkt ausgeführt, an dem der erzeugte Parser (intern) die Stelle erreicht, an der analog in der Grammatik die semantische Aktion definiert wurde. Erlaubt sind diese Aktionen auf der linken Seite unmittelbar vor dem Doppelpunkt (diese wird somit immer ausgeführt), auf der rechten Seite an beliebiger Stelle (die Ausführung hängt von der Produktion ab). In den semantischen Aktionen

ist das Wort `return` nicht erlaubt, da es schon von ANTLR als Schlüsselwort für ein Rückgabeargument belegt ist. Will man dennoch in Java und C++ Methoden mit `return` nutzen, so müssen diese im Abschnitt für eigene Methoden der Grammatik, formuliert werden.

In Listing 30 wird der Wert der Variable `i`, durch die vier semantischen Aktionen, am Ende der Regel, entweder 1 oder -1 betragen.

```
type {int i = 0;} : {i++;} classOrInterfaceType {i++;}
                  | builtInType {i-;} S ;
```

Listing 30: Regel mit semantischen Aktionen

Sollte in der semantischen Aktion auf ein Symbol der Regel zugegriffen werden, so sind dazu spezielle Operatoren nötig. Dadurch wird die semantische Aktion nicht mehr unverändert in den Zielquelltext kopiert, sondern erfährt zuvor einige Ersetzungen. Dazu siehe aber weiter unten.

Semantische Prädikate. Ein Prädikat beschreibt eine Bedingung, die zur Laufzeit erfüllt sein muss, damit der Parsingvorgang fortgesetzt werden kann. Diese Bedingung muss dann `true` oder `false` sein. Im semantischen Prädikat wird die Bedingung in der Zielsprache formuliert. Diese steht in den Grammatiken zwischen geschweiften Klammern mit einem angehängten Fragezeichen (`{ ... }?`). Dabei sollte die Bedingung immer nach der Semantik eines Symbols fragen, wie beispielsweise in Listing 31. Dort wird nach Abarbeitung der Regel überprüft, ob der Text des Tokens `ID` ein Typname ist.

```
decl : "var" ID ":" t:ID {isTypeName(t.getText())}? ;
```

Listing 31: Regel mit validierenden semantischen Prädikat

Bei o. a. Beispiel handelt es sich um ein validierendes semantisches Prädikat. Es muss einer Produktion immer nachgestellt werden und (wie Assertions) eine Exception auslösen, sollte die Bedingung nicht erfüllt sein. In diesem Beispiel muss dies in der Methode `isTypeName` implementiert sein.

Semantische Prädikate können auch zum eindeutig machen von mehrdeutigen Grammatiken verwendet werden. Die beschriebene Syntax in Listing 32 wäre unter LL(k) mit $k < 2$ nichtdeterminiert, da beide Produktionen mit dem Token `ID` beginnen. Die erste repräsentiert eine Deklaration wie `int i` und die zweite eine Zuweisung wie `i = 5`. Mit dem vorangestellten semantischen Prädikat kann jedoch unterschieden werden, welcher Fall zutrifft. Basiert das Token `ID` auf einem Typnamen, dann ist die Bedingung erfüllt und die erste Produktion wird angewand. Basiert es nicht auf einem Typnamen, so handelt es sich um eine Zuweisung und die zweite Produktion findet Verwendung. Durch den Aufruf der Methode `LT(1)` wird das nächste Token abgerufen (also ein Lookahead von 1).

```
stat : {isTypeName(LT(1))}? ID ID ";"          // Deklaration
     |                       ID "=" expr ";" ; // Zuweisung
```

Listing 32: Regel mit semantischem Prädikat

Syntaktische Prädikate. Beim syntaktischen Prädikat formuliert die Bedingung eine zu erkennende Syntax. Zum Festlegen dieser wird ein „daraus folgt“ (`{ ...}=>`) verwendet. In Listing 33 formuliert die Bedingung, dass ein Symbol `list` von einem Gleich (=) gefolgt werden muss, damit diese erfüllt ist. Analog zum validierenden semantischen Prädikat wird dann die erste Produktion gewählt, sonst die zweite.

```
stat:   ( list "=" )=> list "=" list
    |   list ;
```

Listing 33: Regel mit semantischem Prädikat

Ein- & Ausgabeargumente. Jede Regel kann mit Elementen für die Ein- und Ausgabe von Werten erweitert werden. Diese machen i. d. R. nur Sinn, wenn die übergebenen Werte denn benutzt werden. Möglich ist dies als Argument in einer Regelreferenz, oder durch Verwendung in einer semantischen Aktion.

Durch Anhängen des Typs mit einem Variablennamen in eckigen Klammern an den Regelnamen, kann einer Regel ein Eingabewert übergeben werden. Damit eine Regel einen Wert zurückgibt reicht es, dem Regelnamen das Schlüsselwort `returns`, sowie in eckigen Klammern den Ausgabetyp mit Variablennamen, anzuhängen. Im Beispiel aus Listing 34 wird der Eingabewert, per semantischer Aktion, nur an das Ausgabeargument weitergegeben.

```
type[String in] returns [String out] : classOrInterfaceType
                                     | builtInType { out = in } ;
```

Listing 34: Regel mit Ein- und Ausgabe

Zugriff auf Symbole. Alle Symbole in der Grammatik werden zur Laufzeit durch Objekte repräsentiert. Um auf diese während der Laufzeit, in einer semantischen Aktion, zuzugreifen, stellt ANTLR ebenfalls Sprachelemente zur Verfügung.

Soll auf ein Objekt zugegriffen werden, muss das entsprechende Symbol mit einem Label markiert werden. Ein Label ist ein, dem Symbol vorangestellter Identifier mit Doppelpunkt (:). In Listing 35 wird der Text eines Objekts, welches das Symbol `builtInType` repräsentiert, ausgegeben. Auf diese Weise kann dies nur in Parser und Treeparser geschehen.

```
type  :  classOrInterfaceType
      |  x:builtInType  {System.out.println( x.getText() )} ;
```

Listing 35: Ausgabe des Text eines Symbols über ein Label

Soll in einer Aktion auf den Rückgabewert einer Regelreferenz zugegriffen werden (sofern diese einen liefert), muss das Symbol einem Identifier nur per Gleich (=) zugewiesen werden.

Im Lexer kann nur auf das zu erzeugende Token zugegriffen werden, da jede Regel nur eine Tokendefinition zulässt. Dazu wird kein Label benötigt, sondern es reicht ein Dollarzeichen ($) zu Anfang der semantischen Aktion, für den Zugriff. In Listing 36 wird dadurch der Typ des Tokens `WS` (Whitespace) auf `SKIP` gesetzt, damit der Parser dieses später fallen lässt und zum nächsten übergeht.

```
WS    : ( ' ' | '\r' '\n' | '\n' | '\t' ) {$setType(Token.SKIP);}
   ;
```

Listing 36: Setzen des Typs eines Tokens auf SKIP

Im folgenden Abschnitt wird auf Besonderheiten der Regeln der Lexergrammatik näher eingegangen.

E.1.4 Regeln in der Lexergrammatik

Die Regeln des Lexers definieren lediglich die Tokens. Diese stehen auf der linken Seite und die zu erkennenden Terminale auf der rechten Seite der Regeln. Strings müssen zwischen Anführungszeichen stehen und werden vom Lexer als Zeichensequenz interpretiert (Bsp. aus “for“ wird ’f’ ’o’ ’r’).

Auch in der Lexergrammatik können semantische Aktionen definiert werden. Zwar ist dort jede Ausprägung von Javaquelltexten erlaubt, jedoch ist darüber i. d. R. nur die Steuerung des Lexers (wie etwa Behandlung von erst zur Laufzeit erkennbaren Sonderfällen) sinnvoll. Ferner dürfen Lexerregeln keine benutzerdefinierten Exceptions auslösen[24].

```
QUESTION  :  '?'  ;
LPAREN    :  '('  ;
RPAREN    :  ')'  ;
LBRACK    :  '['  ;
RBRACK    :  ']'  ;
FOR       :  "for"  ;
```

Listing 37: Definition von Tokens

[24]Für Parser und Treeparser gilt dies nicht.

E.1.5 Regeln in der Parsergrammatik

In der Parsergrammatik wird in den Regeln die Syntax der zu akzeptierenden Ausdrücke angegeben. Dabei sollten dieselben Tokens wie in der Lexergrammatik benutzt werden.

In der Parsergrammatik sind die Vorkommen der Tokens als Tokenreferenz zu betrachten, d.h. eine Tokenreferenz veranlasst den Lexer die Zeichenfolge, welche das Token symbolisiert, zu erkennen. Die Tokenreferenzen dürfen nur auf der rechten Seite der Regeln verwendet werden und müssen mit einem Großbuchstaben beginnen. Auf der linken Seite stehen die Symbolbezeichner (in ANTLR auch Regelnamen genannt). Kommt ein Symbolbezeichner auf der rechten Seite vor, dann gilt dieser als Regelreferenz (aber nur innerhalb der Parsergrammatik). Symbolbezeichner und -referenzen beginnen mit einem Kleinbuchstaben. Der Symbolbezeichner der ersten Regel ist auch automatisch das Startsymbol[25].

Wird in der Parsergrammatik ein String benutzt, für welchen im Lexer kein Token mit entsprechendem Bezeichner erzeugt wird, so erstellt ANTLR automatisch ein Token für den String, welches mit `LITERAL_` beginnt (Bsp: `enum` in der Parsergrammatik wird dem Token `LITERAL_enum` zugeordnet).

Im Gegensatz zur Lexergrammatik können Regeln der Parsergrammatik benutzerdefinierte Exceptions auslösen. Ferner kann der Parser dazu veranlasst werden einen AST aufzubauen.

Erzeugung des AST. Um den Parser anzuweisen, einen AST aufzubauen muss im Optionsteil seiner Grammatik die Einstellung `buildAST=true` gesetzt werden. Grundsätzlich werden Regelnamen (linke Seite) zu Vaterknoten der Terminale, Token- und Regelreferenzen der rechten Seite. Dabei wird die Reihenfolge von links nach rechts übernommen.

Ferner kann auch eine Tokenreferenz, auf der rechten Seite der Regel, zu einem Vaterknoten befördert werden. Durch Ergänzung dieser mit einem Accent Circonflexe (ˆ) wird der Parser angewiesen alle Elemente zur rechten der Tokenreferenz als Kindknoten anzuhängen. Als Vaterknoten des Vaterknotens fungiert der aus dem Regelnamen resultierende AST-Knoten. Ein Ausrufezeichen (!) hingegen weist den Parser an, keinen entsprechenden AST-Knoten zu erzeugen und dieses Element somit auch nicht mit in den Baum aufzunehmen. Listing 38 verdeutlicht dies anhand von Beispielregeln. Der resultierende AST ist in Abbildung 35 dargestellt.

```
a : b c      ; // a wird Vaterknoten von b & c
b : B!       ; // b wird ein Blatt (keine Kindknoten)
c : d^ e f   ; // c bekommt Kindknoten d, der Vaterknoten von e & f wird
```

Listing 38: Steuern des AST-Aufbaus in der Parsergrammatik

[25]Der Parsingvorgang beginnt mit dem Aufruf der Parsermethode, welche den gleichen Namen wie das Startsymbol trägt.

Abbildung 35: Resultierender AST der aus *Listing 38* resultiert.

E.1.6 Regeln in der Treepasergrammatik

Neben der Erzeugung von Lexern und Parsern bietet ANTLR auch die Erzeugung von Treeparsern (auch Treewalker genannt) an. Grundsätzlich gilt hier das gleiche wie für die Parsergrammatik, allerdings traversiert der Treeparser einen AST anstelle eines Tokenstreams.

Eine Besonderheit der Treeparsergrammtik ist die Raute (#). Eine Raute veranlasst einen Abstieg im Baum. Die Regel in Listing 39 „matcht“ nur einen Knoten `PLUS`, dessen ersten zwei Kinderknoten vom Typ `INT` sind. Dabei muss der Vaterknoten immer über eine Tokenreferenz identifiziert werden - d. h. dieser muss ursprünglich ein Token im Parser zugrundeliegen. Beim schreiben der Treeparsergrammatik muss deshalb bedacht werden mit welchen Strukturen der AST im Parser aufgebaut wurde.

```
expr : #( PLUS INT INT ) ;
```

Listing 39: Regel die einen Subbaum „matcht“

E.1.7 Umsetzung der Grammatik in Javaquelltext

Nicht nur die Grammatiken sollen gut lesbar sein, sondern auch die erzeugten Parser sollen für einen Entwickler möglichst nachvollziehbar sein. Deshalb wurde ANTLR dahingehend optimiert Quelltexte zu erzeugen, die auch ein Entwickler schreiben würde, wenn er einen Parser von Hand programmieren müsste. Schaut man in die erzeugten Lexer- / Parserquelltexte

Abbildung 36: Beispiel-AST der von der Regel aus *Listing 39* erwartet wird.

hinein, kann man leicht nachvollziehen, wie ANTLR die Regeln der Grammatik in Quelltext (hier in Java), umsetzt.

Zu jedem beschriebenen Element folgt ein Beispiel. Auf der linken Seite steht die Grammatik und auf der rechten der daraus erzeugte Quelltext. Das jeweils thematisierte Element ist auf beiden Seiten farblich hervorgehoben.

Operatoren.

Regeln. Jede Regel wird zu einer Methode, die `final` und standardmäßig `public` ist. Dabei dient der Regelname als Methodenbezeichner. Eine Regelreferenz wird zum Aufruf einer Methode und eine Tokenreferenz weist den Lexer mit `match(Token)` an, dieses zu erkennen. Dazu kommen noch Exceptions, die aber erst im nächsten Abschnitt behandelt werden.

Der Übersichtlichkeit halber, werden in den Quelltexten der folgenden Paragraphen die Modifizierer weggelassen. Der Leser sollte aber immer im Hinterkopf behalten, dass diese immer von ANTLR erzeugt werden.

```
a : A ;
```

```
public final void a(){
  match(A);
}
```

Abbildung 37: Umsetzung einer einfachen Regel

Exceptions. Grundsätzlich gibt ANTLR jeder Methode, die aus einer Regel entstammt, mehrere Exceptions mit. Im Lexer sind dies:

- `RecognitionException`

- `CharStreamException`
- `TokenStreamException`

Im Parser sind dies:

- `RecognitionException`
- `TokenStreamException`

Es kann jedoch sein, dass in dieser Methode keinerlei Exceptions ausgelöst werden. In diesem Fall sind o. a. Exceptions in der Methodendeklaration noch aufgeführt, es existiert im Methodenrumpf aber kein throw-Statement. In Abbildung 38 ist dies der Fall, `MyException` muss in einer semantischen Aktion manuell ausgelöst werden.

Der Übersichtlichkeit halber, werden in den Quelltexten der folgenden Paragraphen die `throws`-Klauseln weggelassen. Der Leser sollte aber immer im Hinterkopf behalten, dass diese trotzdem immer von ANTLR erzeugt wird.

```
a throws MyException : A ;
```

```
void a()
throws RecognitionException,
       TokenStreamException,
       MyException
{
  match(A);
}
```

Abbildung 38: Umsetzung einer Regel die eine Exception auslösen kann

Sichtbarkeit einer Regel. Der Sichtbarkeitsmodifizierer wird unverändert als solcher in den Quelltext übernommen. Ist in der Grammatik keiner angegeben, wird standardmässig `public` verwendet.

```
private a : A ;
```

```
private final a()
{
  match(A);
}
```

Abbildung 39: Umsetzung einer Regel mit Sichtbarkeitsmodifizierer

Semantische Aktion. Diese wird unverändert in den Quelltext des Zielprogramms kopiert und muss deshalb auch in der Zielsprache formuliert werden.

```
{int i; i++; int j=i;}          int i;i++;int j=i;
```

Abbildung 40: Umsetzung einer semantischen Aktion

Semantisches Prädikat. Die Bedingung in einem semantischen Prädikat wird als Bedingung in eine `if`-Klausel kopiert. Auch hier muss diese Bedingung deshalb in der Zielsprache formuliert werden. In Abbildung 41 ist die Bedingung mit `LA(1)==ID` kurzschlussverundet. Befindet sich bei einem Lookahead von 1 kein Token `ID`, so tritt keine der beiden Produktionen ein, sondern ein Fehlerfall.

```
stat : {isTypeName(LT(1))}?
         ID ID ";"
     | ID "=" expr ";"
     ;
```

```
if(LA(1)==ID && isTypeName(LT(1))){
    // ID ID ";"
}
else if(LA(1)==ID){
    // ID "=" expr ";"
}
else{
    // Fehler
}
```

Abbildung 41: Umsetzung eines semantischen Prädikats

Eingabe- & Ausgabeargumente. Die Eingabeargumente einer Regel werden zu den Eingabeargumenten der daraus erzeugten Methode. Das Ausgabeargument wird zum Rückgabeargument der Methode. Da eine Methode nur ein Argument zum Rückgabezeitpunkt ausgeben kann, ist auch nur ein Ausgabeargument erlaubt. In Abbildung 42 wird in der Regel `mexpr` der Eingabewert dem Ausgabeargument, per semantischer Aktion, zugewiesen.

```
mexpr[int x] returns [int value=0]
  : ... {value = x;}
  ;
```

```
public int mexpr(int x){
  int value=0;
  ...
  value = x;
  return value;
}
```

Abbildung 42: Umsetzung von Ein- & Ausgabeargumenten

E.2 Ein Beispieltaschenrechner

In diesem Abschnitt sollen einige der zuvor kennengelernten Sprachelemente in einem Beispiel veranschaulicht werden. Dazu soll ein Taschenrechner für arithmetische Ausdrücke, welche nur Addition, Subtraktion, Multiplikation, Klammern und die natürlichen Zahlen verwenden, entwickelt werden. Dabei sollen alle drei Grammatikarten - Lexer, Parser und Treeparser zum Einsatz kommen.

Aus Grund der Übersichtlichkeit Tokens und Tokenreferenzen in allen Grammatiken komplett großgeschrieben, zusätzlich werden Schlüsselworte unterstrichen.

E.2.1 Lexergrammatik

Zunächst wird der Lexer definiert. Für alle zu erkennenden Symbole müssen die jeweils zu erzeugenden Token definiert werden. Neben den o. a. Elementen sollen auch Leerzeichen, Tabulatoren und Zeilenumbrüche verarbeitet werden können. Die Lexergrammatik in Listing 40 soll dies leisten.

```
class ExprLexer extends Lexer;

options {
    k=2; // Lookahead für Zeilenumbrüche
    charVocabulary='\u0000'..'\u007F'; // ASCII
}

LPAREN: '(' ;
RPAREN: ')' ;
PLUS  : '+' ;
MINUS : '-' ;
STAR  : '*' ;
INT   : ('0'..'9')+ ;
WS    : ( ' '
        | '\r' '\n'
        | '\n'
        | '\t'
        )
        {$setType(Token.SKIP);}
      ;
```

Listing 40: Lexergrammatik für Taschenrechner aus *calculator.g*

Nach der letzten Produktion steht eine semantische Aktion. Diese stellt den Typ des Tokens `WS` auf `SKIP`, damit der Parser dieses später fallen lässt und zum nächsten übergeht.

E.2.2 Parsergrammatik

In der Parsergrammatik wird die Syntax der zu akzeptierenden arithmetischen Ausdrücke festgelegt. Es müssen dazu dieselben Tokens wie in der Lexergrammatik benutzt werden.

```
class ExprParser extends Parser;

expr:    mexpr ((PLUS|MINUS) mexpr)*
    ;

mexpr
    :    atom (STAR atom)*
    ;

atom:    INT
    |    LPAREN expr RPAREN
    ;
```

Listing 41: Parsergrammatik für Taschenrechner aus *calculator.g*

Mit Lexer- und Parsergrammatik ist es bereits möglich arithmetische Ausdrücke zu akzeptieren. In diesem Sinne wird im nächsten Abschnitt die Erzeugung und Benutzung von Lexer und Parser besprochen. Auf die Auswertung der Ausdrücke wird anschließend eingegangen.

E.2.3 Erzeugung & Benutzung

Soll der Parser erzeugt werden, muss ANTLR mit den Dateinamen der Grammatiken als Parameter aufgerufen werden. Da beide Grammatiken zusammen in einer Datei stehen können, wird diese in eine Datei `calculator.g` gespeichert. Um Lexer und Parser zu erzeugen reicht ein Aufruf von:

```
java antlr.Tool calculator.g
```

Zuvor muss jedoch die Datei `antlr.jar` in die `CLASSPATH`-Umgebungsvariable aufgenommen werden. ANTLR erstellt dann die Dateien `ExprLexer.java` (enthält den Lexer), `ExprParser.java` (enthält den Parser) und `ExprParserTokenTypes.java` (enthält die gemeinsamen Tokens von Lexer und Parser).

Wie man sieht, werden die Dateien und die teils darin enthaltenen Klassen, wie in den Grammatiken benannt. Um nun den Parser benutzen zu können, wird eine weitere Klasse benötigt, welche die Möglichkeit bereitstellt, arithmetische Ausdrücke einzugeben. Diese müssen dem Lexer bei dessen Instanziierung als Konstruktorparameter übergeben werden, ebenso wie dem Parserkonstruktor anschließend der Lexer als Konstruktorparameter übergeben wird. Der eigentliche Parsingvorgang beginnt mit dem Aufruf der Parsermethode, welche den gleichen

Namen wie das Startsymbol trägt. In unserem Taschenrechner-Beispiel sieht diese Klasse nun wie in Listing 42 aus.

```
import antlr.*;
public class Main {
  public static void main(String[] args) throws Exception {
    //Hier werden die Ausdruecke per Konsole eingegeben
    ExprLexer lexer = new ExprLexer(System.in);
    ExprParser parser = new ExprParser(lexer);
    parser.expr();
  }
}
```

Listing 42: Klasse die dem Benutzer Lexer und Parser zur Verfügung stellt

Nach dem Kompilieren (mit `javac *.java`) und dem Aufruf (mit `java Main`; wobei das ANTLR-Paket noch mit im `CLASSPATH` aufgeführt sein muß), können über die Tastatur arithmetische Ausdrücke eingegeben werden, die dann auf ihre Korrektheit geprüft werden. Ausdrücke wie `3+(4*5)` werden akzeptiert, `3++` hingegen nicht. Eine Auswertung des Ausdrucks erfolgt bisher nicht, dazu jedoch mehr im nächsten Abschnitt.

E.2.4 Auswertung im Parser

Die Auswertung der arithmetischen Ausdrücke erfolgt über semantische Aktionen. Dies kann bereits im Parser geschehen - dazu muss dessen Grammatik wie im Listing 43 aufgeführt, erweitert werden.

```
class ExprParser extends Parser;

expr returns [int value=0]
{int x;}
    :   value=mexpr
        ( PLUS x=mexpr  {value += x;}
        | MINUS x=mexpr {value -= x;}
        )*
    ;

mexpr returns [int value=0]
{int x;}
    :   value=atom ( STAR x=atom {value *= x;} )*
    ;

atom returns [int value=0]
    :   i:INT {value=Integer.parseInt(i.getText());}
```

```
        |    LPAREN value=expr RPAREN
        ;
```

Listing 43: Auswertung des Ausdrucks mit semantische Aktionen in der Parsergrammatik

Die erste Regel gibt das Ergebnis einer Addition oder Subtrakion zurück. Dazu wird die linke Seite der Regel um das Ausgabeargument `returns[int value=0]` und die semantische Aktion `{int x;}` erweitert. Diese deklariert eine lokale Variable, die dazu dient die rechten Operanden der jeweiligen arithmetischen Operation, mit `x=mexpr`, aufzunehmen. Die linken Operanden werden im Ausgabeargument `value` aufgenommen.

Die Auswertung erfolgt schließlich über die semantischen Aktionen `{value += x;}`, wenn eine Addition erkannt und `{value -= x;}`, wenn eine Subtraktion erkannt wird.

Um das Ergebnis auszugeben, muss die Klasse `Main` noch um die in Listing 44 aufgeführten Zeilen ergänzt werden.

```
int x = parser.expr(); // statt nur parser.expr();
System.out.println(x);
```

Listing 44: Angepasste Main-Klasse für Auswertung mittels semantischer Aktionen

Die Eingabe von `3+(4*5)` führt dann zur Ausgabe des Ergebnisses von 23.

Wie oben bereits angeführt, können auch in der Lexergrammatik semantische Aktionen definiert werden. Zwar ist dort jede Ausprägung von Javaquelltexten erlaubt, jedoch ist darüber (im Allgemeinen) nur die Steuerung des Lexers (wie etwa Behandlung von erst zur Laufzeit erkennbaren Sonderfällen) sinnvoll.

E.2.5 Treeparser-Grammatik

Neben der Erzeugung von Lexern und Parsern bietet ANTLR auch die Erzeugung von Treeparsern (auch als Treewalker bezeichnet) an. Grundsätzlich gilt hier das gleiche wie für die Parsergrammatik, allerdings traversiert der Treeparser einen AST anstelle eines Tokenstreams den der Lexer zurückgibt.

Die Auswertung eines arithmetischen Ausdrucks kann auch über eine Treeparser-Grammatik erfolgen.

Bevor der Treeparser zum Einsatz kommen kann, muss jedoch im Optionsteil der Parsergrammatik die Option `buildAST=true` gesetzt werden, um den Parser anzuweisen, auch einen AST aufzubauen. Ferner müssen die Tokens in allen Regeln der Grammatik ergänzt werden, damit der Parser auch weiss, welche Elemente in den AST aufgenommen werden sollen. Die neue Parsergrammatik für den Beispieltaschenrechner sieht dann wie in Listing 45 aus.

Abbildung 43: AST (vereinfacht) für den Ausdruck `3+(4*5)`

```
class ExprParser extends Parser;

options {
        buildAST=true;
}

expr:   mexpr ((PLUS^|MINUS^) mexpr)*
    ;

mexpr
    :   atom (STAR^ atom)*
    ;

atom:   INT
    |   LPAREN! expr RPAREN!
    ;
```

Listing 45: Angepasste Parsergrammatik zur Erzeugung eines AST

Ein Accent Circonflexe (ˆ) hinter einer Tokenreferenz weist den Parser an, einen entsprechenden AST-Knoten und alle weiteren Elemente der Regel als dessen Kinderknoten in den Baum aufzunehmen (von links nach rechts). Also einen Subbaum mit dem aus der Tokenreferenz resultierenden AST-Knoten als Wurzelknoten. Ein Ausrufezeichen (!) hingegen weist den Parser an, keinen entsprechenden AST-Knoten zu erzeugen und auch nicht in den Baum aufzunehmen.

Alle anderen Elemente, d.h. Regelnamen (linke Seite), werden dann automatisch, auch als AST-Knoten, in den Baum aufgenommen. Dabei werden für die Symbole auf der rechten Seite (auch von links nach rechts) entsprechende Kinderknoten angehangen. Die Struktur eines AST für ein einfaches Beispiel ist in Abbildung 43 ersichtlich.

Die Treeparser-Grammatik, welche die arithmetischen Ausdrücke auswertet, beginnt mit einem Optionsteil, der den Treeparser anweist die Tokens des o.a. Lexer und Parser zu übernehmen.

Die Treeparser-Grammatike für das Taschenrechnerbeispiel wird in Listing 46 dargestellt.

```
class ExprTreeParser extends TreeParser;

options {
    importVocab=ExprParser;
}

expr returns [int r=0]
{ int a,b; }
    :   #(PLUS  a=expr b=expr)  {r = a+b;}
    |   #(MINUS a=expr b=expr)  {r = a-b;}
    |   #(STAR  a=expr b=expr)  {r = a*b;}
    |   i:INT {r = (int)Integer.parseInt(i.getText());}
    ;
```

Listing 46: Auswertung des Ausdrucks mit semantische Aktionen in der Treeparser-Grammatik

Eine Besonderheit der Treeparsergrammtik ist die Raute (#). Eine Raute veranlasst einen oder mehrere Abstiege im Baum. Die Regel `#(PLUS expr expr)` „matched“ nur einen Knoten `PLUS`, der zwei Kinderknoten `expr` besitzt. Dabei muss der Vaterknoten immer über eine Tokenreferenz identifiziert werden (d. h. dem muss ursprünglich ein Token zugrundeliegen). Auch hier erfolgt die Auswertung in semantischen Aktionen. Es fällt sofort auf, dass hier nur eine Regel zur Auswertung nötig ist und eine Betrachtung der Präzedenzen entfällt, da diese durch die Struktur des Baumes abgebildet werden.

Die Erzeugung des Treeparsers geschieht analog zur Erzeugung des Lexers und Parsers. ANTLR erzeugt die Klassen `ExprTreeParser.java` und `ExprTreeParserTokenTypes.java`. Benutzt werden kann der Treeparser in o.a. Main-Klasse, allerdings muss diese mit den Zeilen aus Listing 47 ergänzt werden.

```
//davor muss der Parser die Eingabe verarbeitet haben
AST t = parser.getAST();
System.out.println(t.toStringTree());
ExprTreeParser treeParser = new ExprTreeParser();
int x = treeParser.expr(t);
System.out.println(x);
```

Listing 47: Anwenden des Treeparsers auf den AST

Auch hier führt die Eingabe von `3+(4*5)` zum Ergebnis von `23`.

F Entwicklung eines neuen Metamodells

F.1 Vorgehensweise

Als Grundlage des Metamodells dient die Java 5 Grammatik für ANTLR v2.7.6. Zunächst wurden aus der Vorlage von Bodo Hinterwäller allgemeine Vorgehensweisen, um ein Metamodell aus einer Grammatik zu gewinnen, hergeleitet.

Im nächsten Schritt wurde ein Entwurf des Metamodells erstellt, der sich jedoch zu stark an der Ursprungsgrammatik orientierte. So wurde das Metamodell anschließend mehreren Überarbeitungen unterworfen. Im Zuge dieser wurde eine qualitative Verbesserung hauptsächlich durch eine kontinuierliche Vereinfachung des Metamodells erreicht.

Der folgende Abschnitt beschreibt die Heuristiken, welche bei der ersten Fassung des Metamodells, Verwendung fanden.

F.1.1 Heuristiken für den Entwurf

Das Metamodell wird durch ein Klassendiagramm in UML beschrieben. Die vorgestellten Heuristiken beschreiben deshalb die Umsetzung der Elemente der Grammatik in Klassen, Assoziationen, Multiplizitäten etc.

- Klassen:
  - Nichtterminalsymbole werden zu Klassen.
  - Terminalsymbole werden weggelassen.
- Assoziationen:
  - Veroderungen werden zu Generalisierungen. Wobei die Basisklasse auf dem Symbol der linken Seite der Regel fußt.
  - Verkettungen werden zu Aggregationen. Wobei die aggregierende Klasse auf dem Symbol der linken Seite der Regel fußt.
- Multiplizitäten:
  - `(Iteration, Wiederholung) ( Symbol )*` werden zu 0..* und alle (Option) `[ Symbol ]` werden zu 0..1.
  - Symbole ohne o. g. Klammerungen führen zu einer Multiplizität von genau 1.
  - Sind aufgrund komplizierter Grammatikausdrücke mehrere Multiplizitäten möglich, gelten die kleinste untere und die größte obere Grenze.

  - Dabei werden die Multiplizitäten am Ende der Assoziation notiert, welche in der Klasse mündet, die vom Symbol abstammt.

- Rollen
  - Aggregierte Klassen, die von einem Symbol der rechten Regelseite abstammen, sind mit einer Rolle, relativ zur aggregierenden Klasse, zu annotieren.
  - Schlüsselworte werden zu Rollen für die Klassen vor deren Ursprungssymbol sie stehen.

Abbildung 44 und Listing 48 verdeutlichen die Heuristiken an einem Beispiel.

```
a ::= b | c ;
b ::= "foo" ;
c ::= d e { "," e } ;
```

Listing 48: Beispielgrammatik

Abbildung 44: Umsetzung der Grammatik aus Listing 48 in ein Modell.

F.1.2 Entwurf und Überarbeitung des Metamodells

Da die Entwicklung des Metamodells nicht an einem Tag geschah, sondern viele kleine Schritte benötigte, haben wir eine Versionierung eingeführt. Jede u. a. Version des Metamodells entspricht einem Ausdruck, der zu den Treffen den Betreuern vorgelegt wurde.

V1. Nachdem die o. a. Heuristiken möglichst konsequent angewandt wurden, erhielten wir die erste Fassung des Metamodells. Die meisten Regeln und Symbole waren in Generalisierungs- und Aggregationsstrukturen des Diagramms als Klassen vertreten. Auch die ermittelten Multiplizitäten waren in den meisten Fällen zutreffend.

Insgesamt war die Strukur des Metamodells jedoch unübersichtlich und kompliziert und lehnte sich noch zu stark an die Syntax von Java an. Dies lag an der bedenkenlosen Übernahme von möglichst vielen Regeln und Symbolen, um ein möglichst vollständiges Metamodell zu erhalten. Dadurch waren die Präzedenzen der Syntax von Java im Metamodell noch vertreten. Besonders durch die Berücksichtigung der Operatorpräzedenzen führte dies zu komplizierten (verästelten) und unübersichtlichen Strukturen mit uneleganten Bezeichnern (wie `AnnotationDefinitionPart2`). Die nummerierten Klassen enstanden als Behelfskonstrukte bei der Anwendung der Heuristiken zur Veroderung und Verkettung. Die Klassen hatten keinen direkten Vertreter in der Gramatik. Im Nachinein betrachtet sind diese Heuristiken nicht zur Abbildung von Präzenden geeignet.

Zunächst waren die Assoziationen nur als Aggregationen mit Rollen im Modell vertreten. Diese sollten im Zuge der Überarbeitungen durch einfache Assoziationen mit Namen ersetzt werden.

Da alle Bezeichner unverändert aus der Grammatik übernommen worden waren, waren übliche Java-Termini nur wenig vertreten. Ein Großteil der Klassen trug den eigenen Namen als Rollenbezeichner.

Insgesamt bestand das Modell aus 175 Klassen, 247 Aggregationen und 87 Generalisierungen. Eine Überarbeitung des Metamodells war somit dringend erforderlich.

V2. Der Fokus der ersten Überarbeitung lag auf den im vorherigen Abschnitt angeführten Schwächen des Modells. Diese wurden exemplarisch an den Knotentypen `Statement` und `Expression` behoben.

Dazu wurde zunächst von der Syntax und den inhärenten Präzedenzen abstrahiert, um flache Generalisierungshierarchien zu erhalten. Alle Anweisungen waren direkt von `Statement` und alle Ausdrücke direkt von `Expression` abgeleitet. Dann wurden alle Operatoren in den Klassen `PrefixExpression`, `InfixExpression` und `PostfixExpression` als Enumerationen zusammengefasst (siehe Abbildungen 2 und 3). Insgesamt sorgte dies bereits für ein übersichtlicheres Modell.

Anschließend wurde nach aussagekräftigen Bezeichnern für Klassen und den noch vertretenen Rollen gesucht. Danach waren erheblich mehr Java-Termini im Metamodell vertreten und die Namen bezeichnender.

Abbildung 45: Modell der `PostFixExpression` aus dem Entwurf.

Das Modell war „nur“ zu einem Drittel überarbeitet, jedoch dadurch schon signifikant kleiner und überschaubarer. Das Klassendiagramm bestand nun aus 132 Klassen, 172 Aggregationen und 73 Generalisierungen.

Abbildung 46: Modell der `PostFixExpression` nach Überarbeitung

V3. Im nächsten Schritt haben wurden die oben angeführten Verfahren auf die Knotentypen `ClassDefinition` und `ConstructorDeclaration`. Danach waren die einzelnen Elemente der Klassendefinition und Konstruktordeklaration auf einer Ebene angeordnet. Die syntaktische Reihenfolge kann im Metamodell nicht explizit festgelegt werden, deshalb wurden die Reihenfolge im Klassendiagramm graphisch abgebildet[26]. So kann ein Entwickler, die vom Programmieren her bekannten Strukturen einfacher wiedererkennen.

Der Knotentyp `Type` (später zu `TypeSpecification` umbenannt), der eine Typspezifikation repräsentiert, musste regelrecht zusammengesucht werden, da seine Teile über das ganze Modell verstreut lagen. Hier wählten wir eine Strukturierung in Anlehnung an das Metamodell von Hinterwäller.

Ferner wurden alle Klassen- und Attributnamen im Modells in Anlehnung an Javatermini vereinheitlicht sowie die Knotentypen `EmptyStatement` und `NullExpression` eingeführt. Diese waren in der Grammatik nicht explizit vorhanden, da der Parser die Tokens, welche eine leere Anweisung oder den `null`-Ausdruck repräsentieren, fallen lässt.

Entfernt wurden Klassen die Sammlungen repräsentieren (Bsp. `ParameterList`, `ArgList`). Diese resultieren aus Regeln der Parsergrammatik, die dort für mehr Überschaubarkeit sorgen. Im Metamodell ist das Gegenteil der Fall. Daher wurden diese durch entsprechende Kardinalitäten an den, in den Sammlungen, zusammengefassten Klassen ersetzt.

[26]Für die richtige Reihenfolge der Elemente im Graphen muss der Faktenextraktor sorgen. Er muss sicherstellen, dass die syntaktische Reihenfolge bei der Erstellung des Graphen miteinfliesst, da die einzelnen Elemente durchnummeriert werden.

Abbildung 47: Modell der `TraditionalForClause` aus dem Entwurf.

Nach diesem Schritt war das Metamodell zur Hälfte überarbeitet worden. Es war abermals überschaubarer und dadurch verständlicher geworden. Das Klassendiagramm bestand mittlerweile aus 115 Klassen, 163 Aggregationen und 73 Generalisierungen.

Nun war klar, dass eine generelle Verbesserung des Modells durch eine konsequente Vereinfach erreicht werden konnte.

F.1.3 Weitere Heuristiken

Im Zuge der Überarbeitung konnten weitere Heuristiken zur Erstellung eines Metamodells hergeleitet werden. Diese ergänzen jene aus Abschnitt F.1.1 und erhöhen, angewendet, erheblich die Qualität eines Metamodells:

- strukturell vereinfachend:
  - Verwerfung der Präzedenzen durch Aufbrechen der Strukturen und Gruppierung aller Subklassen einer Klasse in flachen Generalisierungen. Beispiel siehe Abbildung 45 und 46.
  - Verkürzung der Wege im Diagramm durch Verwerfung von Klassen, die Sammlungen repräsentieren. Ersetzung dieser mit entsprechenden Kardinalitäten an den vorher, in den Sammlungen, zusammengefassten Klassen. Beispiel siehe Abbildung 47 und 48.
- treffender bezeichnend:
  - Vergabe von aussagekräftigen Bezeichnern in Anlehnung an die allgemeinen Javaprogrammiertermini und die, in der Vorlesung „Programmierung“, gelehrten Regeln. Z. B. die Kantennamen in Abbildung 49.

Abbildung 48: Überarbeitetes Modell des `TraditionalForClause` nach Heuristiken aus Abschnitt F.1.3.

Abbildung 49: Überarbeitetes Modell des `DoWhileStatement` nach Heuristiken aus Abschnitt F.1.3.

F.1.4 Weitere Überarbeitung

V4. Im nächsten Schritt wurden fast alle übrigen Klassen unter konsequenter Anwendung oben angeführter Heuristiken überarbeitet. Hinzu kam, dass im gesamten Metamodell alle Aggregationen durch einfache Assoziationen mit Adjektiven als Kantennamen ersetzt werden mussten. Dies war nötig für die spätere Bearbeitung der resultierenden TGraphen. Die Heuristiken aus Abschnitt F.1.1, welche die Rollen betreffen, sind somit überflüssig.

Eine direkte Umsetzung in einfache Assoziationen mit Kantennamen ist empfehlenswert, da das Metamodell dadurch noch weiter vereinfacht werden kann. Einige Knotentypen werden dadurch unnötig. Zum Beispiel stammen Klassen wie `ImplementsClause` von Regeln der Grammatik ab, die zur Wiederverwendung in anderen Regeln geschrieben wurden. Klassen

und Enums können Interfaces implementieren. In der Grammatik wird somit an zwei Stellen die Regel `ImplementsClause` referenziert. In der Regel selbst wird jedoch nur *eine* weitere Regel referenziert. Diese Referenz kann durch eine einfache Assoziation mit Kantennamen `IsInterfaceOf` ausgedrückt und der Knotentyp `ImplementsClause` somit überflüssig werden. Die Abbildungen 50 und 51 verdeutlichen dies.

Zusätzlich wurden in das Metamodell die Klassen integriert, welche das Navigieren der TGraphen mit Gupro ermöglichen (`Program`, `TranslationUnit` usw.).

Das Metamodell bestand nun aus 89 Klassen, 134 einfachen Assoziationen mit Namen und 63 Generalisierungen.

Abbildung 50: Modell der `ClassDefinition` mit Rollennamen aus dem Entwurf.

Abbildung 51: Modell der `ClassDefinition` mit Kantenname nach Überarbeitung.

V5. Im nächsten Schritt wurden die letzen Klassen überarbeitet. Zusätzlich wurden noch Knoten- und Kantentypen zur Repräsentation von Kommentaren im Quelltext in das Metamodell aufgenommen. Somit war das komplette Metamodell einmal überarbeitet worden. Fast

alle Elemente waren in Generalisierungstrukturen angeordnet. Es konnten dieser sieben unterschieden werden - mit folgenden Klassen als Basistypen[27]:

- *Type*
- *TypeSpecification*
- *Member*
- *Statement*
- *Expression*
- *Annotation*
- *Comment*

Im Zuge der weiteren Überarbeitung sollten diese Klassen später in eigenen Packages geordnet werden. Insgesamt änderte sich gegenüber der Vorversion an der Größe des Metamodells fast nichts. Nur die Anzahl der Knotentypen verringerte sich. Dies lag am doppelten Vorkommen vieler Assoziationsnamen, ein Umstand der in der nächsten Überarbeitung korrigiert werden musste. Trotzdem war das Metamodell übersichtlicher und somit verständlicher geworden. Es bestand somit aus 88 Klassen, 93 einfachen Assoziationen mit Namen und 63 Generalisierungen.

V6. Aufgrund von technischen Restriktionen können Kanten immer nur zwischen zwei fest definierten Knoten existieren. Durch Ableitung der Knoten kann die selbe Kante auch zwischen den abgeleiteten Knoten existieren. Das Metamodell betand aus 89 Klassen, 57 Generalisierungen und 143 einfachen Assoziationen mit Namen.

V7. Im letzten Schritt wurden semantische Kantentypen eingeführt, die keine Attribute haben. Die Kantentypen haben keine Attribute, da sie keinen syntaktischen Ursprung haben. Sie werden dazu verwendet die Graphrepräsentation der Definition eines Elements mit der Graphrepräsentation der Benutzung dieses Elements zu verbinden. Z. B. verbindet die Kante vom Typ `IsTypeDefinitionOf` eine `ClassDefinition` mit einem `QualifiedType`, der eine Typspezifikation repräsentiert. Das Metamodell besteht in seiner finalen Fassung aus 89 Knoten- und 160 Kantentypen.

[27] Diese wurden als abstrakt deklariert.

Abbildung 52: Modell der `TypeParameterDeclaration`.

F.1.5 Verifikation des Metamodells

Wie bereits oben beschrieben war eine immerwährende Überprüfung der Ergebnisse nötig. Anfangs waren es komplizierte Strukturen im Modell, die es zu überprüfen galt. Diese waren meist korrekt, da sie sich noch sehr stark an die Grammatik anlehnten. Im Zuge der Vereinfachung des Modells galt es zu überprüfen, ob die einfachen Strukturen mindestens die selben syntaktischen Konstruktionen aus Java abbildeten, wie ihre komplizierten Pendants.

Deshalb haben wir, zu jedem Subgraphen im Metamodell, Stellvertreter erlaubter syntaktischen Konstruktionen aus Java aufgeschrieben und geprüft ob diese abgedeckt werden. Um diese Überprüfung einfacher vornehmen zu können, haben wir die Elemente im Klassendiagramm so gruppiert, dass sie sich, wie der Quelltext, von Links nach Rechts, lesen lassen. Beispielsweise deckt das Modell aus Abbildung 52 die syntaktischen Konstruktionen aus Listing 49 ab.

```
MyClass < T > { ... }
MyClass < T extends String > { ... }
MyClass < T super YourClass >{ ... }
```

Listing 49: Beispiele für die drei erlaubten syntaktischen Konstruktionen (rotmarkiert) einer `TypeParameterDeclaration`

F.1.6 Designentscheidungen

Vereinheitlichung. Ziel der Designentscheidungen war es, neben der weiteren Vereinfachung des Modells, auch eine Vereinheitlichung der Strukturen zu erreichen, die das Verständnis beim Arbeiten mit den TGraph-Instanzen fördern soll. Vereinheitlichungen nahmen wir vor mit:

- Aufhebung der Unterscheidung der Rümpfe von `Class`, `Interface`, `Enumeration` und `Annotation`. So war eine Zusammenfassung der möglichen Inhalte eines `Block` möglich und das Metamodell wurde erheblich kompakter.
- Nichtunterscheidung des Destructors, da dieser nur eine speziell benannte Methode ist und einfach über seinen Namen *finalize* identifiziert werden kann.
- Einführung von `QualifiedName` analog zu, bereits bestehendem, `QualifiedType`. So können diese, z. B. bei GreQL-Anfragen, auch auf die gleiche Weise behandelt werden.

Eindeutig unterschieden haben wir dagegen Deklarationen und Definitionen von Fields, lokalen Variablen und Methoden, da in Java nicht generell Deklaration und Definition, wie in C++, getrennt geschehen.

Semantik. In das Metamodell sind auch Designentscheidungen eingeflossen, die nicht direkt das Modell, sondern den Javaextraktor an sich betreffen, da sie nicht vollständig, durch die Semantik des Metamodells, abgedeckt werden. Dazu zählen:

- Behandlung des „super“ eines Aufrufs einer Supermethode (z. B. `super.toString()`), als `Identifier`, der, über eine Kante `isIdentifierOf`, mit der Superklasse verbunden ist.
- „Höherbewertung“ des `Package` als `TranslationUnit`, da in Java ein Programm ein Package sein sollte, welches wiederum aus Packages bestehen kann.

Attribute. Durch Attribute konnten, unter anderem, viele unnötigen Klassen eingespart werden, die aus syntaktische Einschränkungen der Grammatik stammten. Folgende Attribute haben wir eingeführt:

- String-Attribute ordnen den Klassen ihre Bezeichner zu:
  - *name* in `Identifier`.
  - *fullyQualifiedName* in
    - `QualifiedName` und seiner Subklasse
    - `Type` und seinen Subklassen (z. B. `ClassDefinition`).
    - `JavaPackage`
- Integer-Attribute ordnen den Klassen ihren Ursprung, aus Java, in einer Enumeration, zu:
  - *type* in `Modifier`.

  - *type* in `BuiltInType`.
  - *prefixOperator* in `PrefixExpression`, analog für `InfixExpression` und `PostfixExpression`.
- Boolean-Attribute ersetzen Generalisierungen:
  - *superMethod* und *constructorMethod* in `MethodInvocation`.

Die `StringConstantExpression` besitzt ein Attribut *value* vom Typ String, der den Wert dieses Ausdrucks enthält. Analog verhält es sich bei den konstanten Ausdrücken für die Typen Boolean, Integer, Long, Float, Double und Character.

Generell sind viele syntaktische Einschränkungen, der Ursprungsgrammatik, in unserem Metamodell nicht mehr vertreten. Durch die Vereinfachung des Modells kam es zur Aufweichung der Einschränkungen. Es erlaubt mehr syntaktische Konstruktionen als die Grammtik - mitunter sogar falsche. Diese Falschen zu identifizieren, ist jedoch Aufgabe des Parsers.

F.2 Javagrammatik aus ANTLR

Folgenden Grammatik stammt in von Michael Studmann und ist für Java 5 ausgelegt. Um diese lesbarer zu machen, haben wir sie von Kommentaren, semantischen Aktionen usw. bereinigt und in eine EBNF-konforme Syntax überführt. Ferner wurden in allen Regeln die Tokenreferenzen durch das ursprüngliche Terminalsymbol ersetzt. Diese Grammatik stellt somit eine (vereinfachte) Verschmelzung von Lexer- und Parsergrammatik dar.

```
1  compilationUnit
2          ::=     [ packageDefinition ]
3                  { importDefinition }
4                  { typeDefinition }
5                  EOF
6          ;
7
8  packageDefinition
9          ::=     annotations "package" identifier ";"
10         ;
11
12 importDefinition
13         ::=     "import" [ "static" ] identifierStar ";"
14         ;
15
16 typeDefinition
17         ::=     modifiers
18                 typeDefinitionInternal
19         |       ";"
```

```
20              ;
21
22  typeDefinitionInternal
23              ::=      classDefinition
24              |        interfaceDefinition
25              |        enumDefinition
26              |        annotationDefinition
27              ;
28
29  declaration
30              ::=      modifiers typeSpec variableDefinitions
31              ;
32
33  typeSpec
34              ::=      classTypeSpec | builtInTypeSpec
35              ;
36
37  classTypeSpec
38              ::=      classOrInterfaceType
39                       { "[" "]" }
40              ;
41
42  classOrInterfaceType
43              ::=      IDENT [typeArguments]
44                       { "." IDENT [typeArguments] }
45              ;
46
47  typeArgumentSpec
48              ::=      classTypeSpec | builtInTypeArraySpec
49              ;
50
51  typeArgument
52              ::=      typeArgumentSpec | wildcardType
53              ;
54
55  wildcardType
56              ::=      "?" [typeArgumentBounds]
57              ;
58
59  typeArguments
60              ::=      "<"
61                       typeArgument
62                       { "," typeArgument }
63                       [ typeArgumentsOrParametersEnd ]
```

```
64          ;
65
66  typeArgumentsOrParametersEnd
67          ::=     ">"
68          |       ">>"
69          |       ">>>"
70          ;
71
72  typeArgumentBounds
73          ::=     ( "extends" | "super" ) classOrInterfaceType
74          ;
75
76  builtInTypeArraySpec
77          ::=     builtInType "[" "]" { "[" "]" }
78          ;
79
80  builtInTypeSpec
81          ::=     builtInType { "[" "]" }
82          ;
83
84  type
85          ::=     classOrInterfaceType | builtInType
86          ;
87
88  builtInType
89          ::=     "void"
90          |       "boolean"
91          |       "byte"
92          |       "char"
93          |       "short"
94          |       "int"
95          |       "float"
96          |       "long"
97          |       "double"
98          ;
99
100 identifier
101         ::=     IDENT { "." IDENT }
102         ;
103
104 identifierStar
105         ::=     IDENT
106                 { "." IDENT }
107                 [ "." "*" ]
```

```
108            ;
109
110 modifiers
111            ::=      { modifier | annotation }
112            ;
113
114 modifier
115            ::=      "private"
116            |        "public"
117            |        "protected"
118            |        "static"
119            |        "transient"
120            |        "final"
121            |        "abstract"
122            |        "native"
123            |        "threadsafe"
124            |        "synchronized"
125            |        "volatile"
126            |        "strictfp"
127            ;
128
129 annotation
130            ::=      "@" identifier [ "(" [ annotationArguments ] ")" ]
131            ;
132
133 annotations
134            ::=    {annotation}
135            ;
136
137 annotationArguments
138            ::=      annotationMemberValueInitializer |
                  anntotationMemberValuePairs
139            ;
140
141 anntotationMemberValuePairs
142            ::=      annotationMemberValuePair { ","
                  annotationMemberValuePair }
143            ;
144
145 annotationMemberValuePair
146            ::=      IDENT "=" annotationMemberValueInitializer
147            ;
148
149 annotationMemberValueInitializer
```

```
150         ::=
151                 conditionalExpression | annotation |
                        annotationMemberArrayInitializer
152         ;
153 
154 annotationMemberArrayInitializer
155         ::=     "{"
156                 [ annotationMemberArrayValueInitializer
157                   { "," annotationMemberArrayValueInitializer }
158                   [ "," ]
159                 ]
160                 "}"
161         ;
162 
163 annotationMemberArrayValueInitializer
164         ::=     conditionalExpression
165         |       annotation
166         ;
167 
168 superClassClause
169         ::=     [ "extends" classOrInterfaceType ]
170         ;
171 
172 classDefinition
173         ::=     "class" IDENT
174                 [ typeParameters ]
175                 superClassClause
176                 implementsClause
177                 classBlock
178         ;
179 
180 interfaceDefinition
181         ::=     "interface" IDENT
182                 [ typeParameters ]
183                 interfaceExtends
184                 interfaceBlock
185         ;
186 
187 enumDefinition
188         ::=     "enum" IDENT
189                 implementsClause
190                 enumBlock
191         ;
192 
```

```
193 annotationDefinition
194         ::=     "@" "interface" IDENT
195                 annotationBlock
196         ;
197 
198 typeParameters
199         ::=
200                 "<"
201                 typeParameter { "," typeParameter }
202                 [ typeArgumentsOrParametersEnd ]
203         ;
204 
205 typeParameter
206         ::=
207                 IDENT [ typeParameterBounds ]
208         ;
209 
210 typeParameterBounds
211         ::=
212                 "extends" classOrInterfaceType
213                 { "&" classOrInterfaceType }
214         ;
215 
216 classBlock
217         ::=     "{"
218                         { classField | ";" }
219                 "}"
220         ;
221 
222 interfaceBlock
223         ::=     "{"
224                         { interfaceField | ";" }
225                 "}"
226         ;
227 
228 annotationBlock
229         ::=     "{"
230                         { annotationField | ";" }
231                 "}"
232         ;
233 
234 enumBlock
235         ::=     "{"
```

```
236                                [ enumConstant { "," enumConstant } [ ","
                                      ] ]
237                                [ ";" { classField | ";" } ]
238                      "}"
239            ;
240
241  annotationField
242            ::=       modifiers
243                      (         typeDefinitionInternal
244                      |         typeSpec
245                                (          IDENT
246                                           "(" ")"
247                                           declaratorBrackets
248                                           [ "default"
                                                annotationMemberValueInitializer
                                                 ]
249                                           ";"
250                                |          variableDefinitions ";"
251                                )
252                      )
253            ;
254
255  enumConstant
256            ::=       annotations
257                      IDENT
258                      [ "(" argList ")" ]
259                      [ enumConstantBlock ]
260            ;
261
262  enumConstantBlock
263            ::=       "{" { enumConstantField | ";" } "}"
264            ;
265
266  enumConstantField
267            ::=       modifiers
268                      (         typeDefinitionInternal
269                      |         [ typeParameters ] typeSpec
270                                (          IDENT
271                                           "(" parameterDeclarationList ")"
272                                           declaratorBrackets
273                                           [ throwsClause ]
274                                           ( compoundStatement | ";" )
275                                |          variableDefinitions ";"
276                                )
```

```
277                 )
278         |       compoundStatement
279         ;
280
281 interfaceExtends
282         ::=     [ "extends" classOrInterfaceType { ","
              classOrInterfaceType } ]
283         ;
284
285 implementsClause
286         ::=     [ "implements" classOrInterfaceType { ","
              classOrInterfaceType } ]
287         ;
288
289 classField
290         ::=     modifiers
291                 (       typeDefinitionInternal
292                 |       (typeParameters]
293                         (
294                                 ctorHead constructorBody
295                         |       typeSpec
296                                 (       IDENT
297                                         "("
                                              parameterDeclarationList
                                               ")"
298                                         declaratorBrackets
299                                         [ throwsClause ]
300                                 |       variableDefinitions ";"
301                                 )
302                         )
303                 )
304         |       [ "static" ] compoundStatement
305         ;
306
307 interfaceField
308         ::=     modifiers
309                 (       typeDefinitionInternal
310                 |       [ typeParameters ]
311                         typeSpec
312                         (       IDENT
313                                 "(" parameterDeclarationList ")"
314                                 declaratorBrackets
315                                 [ throwsClause ]
316                                 ";"
```

```
317                             |       variableDefinitions ";"
318                             )
319                     )
320             ;
321
322 constructorBody
323             ::=     "{" [ explicitConstructorInvocation ] {statement}
                  "}"
324             ;
325
326 explicitConstructorInvocation
327             ::=     [ typeArguments ]
328                     (       "this" "(" argList ")" ";"
329                     |       "super" "(" argList ")" ";"
330                     )
331             ;
332
333 variableDefinitions
334             ::=     variableDeclarator { "," variableDeclarator }
335             ;
336
337 variableDeclarator
338             ::=     IDENT declaratorBrackets varInitializer
339             ;
340
341 declaratorBrackets
342             ::=     { "[" "]" }
343             ;
344
345 varInitializer
346             ::=     [ "=" initializer ]
347             ;
348
349 arrayInitializer
350             ::=     "{"
351                             [       initializer
352                                     { "," initializer }
353                                     [","]
354                             ]
355                     "}"
356             ;
357
358
359 initializer
```

```
360         ::=     expression | arrayInitializer
361         ;
362
363 ctorHead
364         ::=     IDENT
365                 "(" parameterDeclarationList ")"
366                 [ throwsClause ]
367         ;
368
369 throwsClause
370         ::=     "throws" identifier { "," identifier }
371         ;
372
373 parameterDeclarationList
374         ::=     [       parameterDeclaration
375                         { "," parameterDeclaration }
376                         [ "," variableLengthParameterDeclaration ]
377                 |
378                         variableLengthParameterDeclaration
379                 ]
380         ;
381
382 parameterDeclaration
383         ::=     parameterModifier typeSpec IDENT
                declaratorBrackets
384         ;
385
386 variableLengthParameterDeclaration
387         ::=     parameterModifier typeSpec "..." IDENT
                declaratorBrackets
388         ;
389
390 parameterModifier
391         ::=     { annotation } [ "final" ] { annotation }
392         ;
393
394 compoundStatement
395         ::=     "{" { statement } "}"
396         ;
397
398 statement
399         ::=     compoundStatement
400         |       declaration ";"
401         |       expression ";"
```

```
402         |       modifiers classDefinition
403         |       IDENT ":" statement
404         |       "if" "(" expression ")" statement [ "else"
                  statement ]
405         |       forStatement
406         |       "while" "(" expression ")" statement
407         |       "do" statement "while" "(" expression ")" ";"
408         |       "break" [ IDENT ] ";"
409         |       "continue" [ IDENT ] ";"
410         |       "return" [ expression ] ";"
411         |       "switch" "(" expression ")" "{" { casesGroup } "}"
412         |       tryBlock
413         |       "throw" expression ";"
414         |       "synchronized" "(" expression ")"
                  compoundStatement
415         |       "assert" expression [ ":" expression ] ";"
416         |       ";"
417         ;
418
419 forStatement
420         ::=     "for" "(" ( traditionalForClause | forEachClause )
                  ")" statement
421         ;
422
423 traditionalForClause
424         ::=     forInit ";" forCond ";" forIter
425         ;
426
427 forEachClause
428         ::=     parameterDeclaration ":" expression
429         ;
430
431 casesGroup
432         ::=     aCase { aCase } caseSList
433         ;
434
435 aCase
436         ::=     ("case" expression | "default") ":"
437         ;
438
439 caseSList
440         ::=     { statement }
441         ;
442
```

```
443 forInit
444         ::=     [ declaration | expressionList ]
445         ;
446
447 forCond
448         ::=     [ expression ]
449         ;
450
451 forIter
452         ::=     [ expressionList ]
453         ;
454
455 tryBlock
456         ::=     "try" compoundStatement { handler } [
                finallyClause ]
457         ;
458
459 finallyClause
460         ::=     "finally" compoundStatement
461         ;
462
463 handler
464         ::=     "catch" "(" parameterDeclaration ")"
                compoundStatement
465         ;
466
467 expressionList
468         ::=     expression { "," expression }
469         ;
470
471 expression
472         ::=     assignmentExpression
473         ;
474
475 assignmentExpression
476         ::=     conditionalExpression
477                 [       (       "="
478                         |       "+="
479                         |       "-="
480                         |       "*="
481                         |       "/="
482                         |       "%="
483                         |       ">>="
484                         |       ">>>="
```

```
485                     |       "<<="
486                     |       "&="
487                     |       "^="
488                     |       "|="
489                     )
490                     assignmentExpression
491                 ]
492         ;
493
494 conditionalExpression
495         ::=     logicalOrExpression [ "?" assignmentExpression ":"
              conditionalExpression ]
496         ;
497
498 logicalOrExpression
499         ::=     logicalAndExpression { "||" logicalAndExpression }
500         ;
501
502 logicalAndExpression
503         ::=     inclusiveOrExpression { "&&" inclusiveOrExpression
              }
504         ;
505
506 inclusiveOrExpression
507         ::=     exclusiveOrExpression { "|" exclusiveOrExpression
             }
508         ;
509
510 exclusiveOrExpression
511         ::=     andExpression { "^" andExpression }
512         ;
513
514 andExpression
515         ::=     equalityExpression { "&" equalityExpression }
516         ;
517
518 equalityExpression
519         ::=     relationalExpression { ( "!=" | "==" )
             relationalExpression }
520         ;
521
522 relationalExpression
523         ::=     shiftExpression
524                 (       {       (       "<"
```

```
525                                    |          ">"
526                                    |          "<="
527                                    |          ">="
528                                    )
529                                    shiftExpression
530                           }
531                  |        "instanceof" typeSpec
532                  )
533         ;
534
535 shiftExpression
536         ::=      additiveExpression { ( "<<" | ">>" | ">>>" )
            additiveExpression }
537         ;
538
539 additiveExpression
540         ::=      multiplicativeExpression { ( "+" | "-" )
            multiplicativeExpression }
541         ;
542
543 multiplicativeExpression
544         ::=      unaryExpression { ( "*" | "/" | "%" )
            unaryExpression }
545         ;
546
547 unaryExpression
548         ::=      "++" unaryExpression
549         |        "--" unaryExpression
550         |        "-" unaryExpression
551         |        "+" unaryExpression
552         |        unaryExpressionNotPlusMinus
553         ;
554
555 unaryExpressionNotPlusMinus
556         ::=      "~" unaryExpression
557         |        "!" unaryExpression
558         |        "(" builtInTypeSpec ")" unaryExpression
559         |        "(" classTypeSpec ")" unaryExpressionNotPlusMinus
560         |        postfixExpression
561         ;
562
563 postfixExpression
564         ::=      primaryExpression
565                  {        "." [ typeArguments ]
```

```
566                                       (         IDENT [ "(" argList ")" ]
567                                       |         "super"
568                                                 (        "(" argList ")"
569                                                 |        "." [
                                                       typeArguments ] IDENT [
                                                        "(" argList ")" ]
570                                                 )
571                                       )
572                    |        "." "this"
573                    |        "." newExpression
574                    |        "[" expression "]"
575                    }
576                    [ "++" | "--" ]
577           ;
578
579 primaryExpression
580           ::=      identPrimary [ "." "class" ]
581           |        constant
582           |        "true"
583           |        "false"
584           |        "null"
585           |        newExpression
586           |        "this"
587           |        "super"
588           |        "(" assignmentExpression ")"
589           |        builtInType { "[" "]" } "." "class"
590           ;
591
592 identPrimary
593           ::=      [ typeArguments ]
594                    IDENT
595                    { "." [ typeArguments ] IDENT }
596                    [        "(" argList ")"
597                    |        "[" "]" { "[" "]" }
598                    ]
599           ;
600
601 newExpression
602           ::=      "new" [ typeArguments ] type
603                    (        "(" argList ")" [ classBlock ]
604                    |        newArrayDeclarator [ arrayInitializer ]
605                    )
606           ;
607
```

```
608 argList
609          ::=        expressionList | /* nothing */
610          ;
611
612 newArrayDeclarator
613          ::=        "[" [ expression ] "]" { "[" [ expression ] "]" }
614          ;
615
616 constant
617          ::=        NUM_INT
618          |          CHAR_LITERAL
619          |          STRING_LITERAL
620          |          NUM_FLOAT
621          |          NUM_LONG
622          |          NUM_DOUBLE
623          ;
```

Listing 50: Beispiele für die drei erlaubten syntaktischen Konstruktionen (rotmarkiert) einer `TypeParameterDeclaration`

Literatur

[1] Aho, Alfred V.; Ullman, Jeffrey D. (1977):
Principles of Compiler Design
Dritte Auflage; Reading, Massachusetts: Addison-Wesley, 1979. ISBN 0-201-00022-9

[2] Bildhauer, Daniel (2006):
QGuPro Sourcecodebrowser mit Unterstützung von Folding, Syntaxhighlighting und graphbasierter Navigation.
Zugl.: Koblenz, Univ., Studienarbeit, 2006

[3] Bildhauer, Daniel (2006):
Ein Interpreter für GReQL 2: Entwurf und prototypische Implementation.
Zugl.: Koblenz, Univ., Dipl., 2006

[4] Bornstein, Dan (2001):
ANTLR Adder Tutorial: Extent Tracking, Tokens with Values, and Error Reporting (Version 1.3)
http://www.milk.com/kodebase/antlr-tutorial/
Abruf: 01/2008

[5] Bracha, Gilad (2005):
The Java Language Specification, Third Ed.
Sun Microsystems, 2005.
http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html
Abruf: 01/2008

[6] Ebert, Jürgen; Gimnich, Rainer; Stasch, Hans H.; Winter, Andreas (1998):
Gupro: Generische Umgebung zum Programmverstehen.
Koblenz: Fölbach, 1998. ISBN 3-923532-59-8

[7] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995):
Design Patterns: Objects of Reusable Object-Oriented Software.
Sechste Auflage; Reading, Massachusetts: Addison-Wesley, 1996. ISBN 0-201-63361-2

[8] Gosling, James; Joy, Bill; Steele, Guy (1996):
The Java Language Specification, First Ed.
Sun Microsystems, 1996.
http://java.sun.com/docs/books/jls/first_edition/html/index.html
Abruf: 01/2008

[9] Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad (2000):
The Java Language Specification, Second Ed.
Sun Microsystems, 2000.
http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html
Abruf: 01/2008

[10] Hinterwäller, Bodo (2005):
Metamodell basierte Spezifikation von Refactorings.
Zugl.: Koblenz, Univ., Dipl., 2005

[11] Kahle, Steffen (2006):
JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen.
Zugl.: Koblenz, Univ., Dipl., 2006

[12] Marchewka, Katrin (2006):
Entwurf und Definition der Graphanfragesprache GReQL 2.
Zugl.: Koblenz, Univ., Dipl., 2006

[13] Parr, Terrence:
ANTLR Website.
http://www.antlr.org/
Abruf: 01/2008

[14] Parr, Terrence:
An Introduction To ANTLR.
http://www.cs.usfca.edu/~parrt/course/652/lectures/antlr.html
Abruf: 09/2006

[15] Parr, Terrence (2005):
ANTLR Reference Manual.
http://www.antlr.org/doc/index.html
Abruf: 09/2006

[16] Parr, Terrence:
ANTLR Reference Manual: Grammar Options.
http://www.antlr.org/doc/options.html
Abruf: 09/2006

[17] Parr, Terrence:
ANTLR Reference Manual: Meta Language.
http://www.antlr.org/doc/metalang.html#_bb2
Abruf: 09/2006

[18] Studman, Michael:
Java 5 Grammar for ANTLR.
http://www.antlr.org/grammar/1090713067533/index.html
Abruf: 01/2008

[19] *Gupro: Re-Group Projekt SVN Repository*
https://svn.uni-koblenz.de/gup/re-group/trunk/project/
Abruf: 09/2006

[21] *Homepage von JLex*.
http://www.cs.princeton.edu/~appel/modern/java/JLex/
Abruf: 01/2008

[22] *Grammatiken für JavaCC*.
http://javacc.dev.java.net/
Abruf: 01/2008

[23] *Grammatiken für JavaCC*.
http://www.cobase.cs.ucla.edu/pub/javacc/
Abruf: 06/2006

[24] *CUP bis 1999*.
http://www.cs.princeton.edu/~appel/modern/java/CUP/
Abruf: 01/2008

[25] *CUP ab 1999*.
http://www2.cs.tum.edu/projects/cup/
Abruf: 01/2008

[26] *Homepage von CoCo/R*.
http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/
Abruf: 01/2008

[27] *Homepage von CoCoLab*.
http://www.cocolab.org/
Abruf: 01/2008

[28] *Homepage von JAbstract*.
http://www.doc.gold.ac.uk/~mas01sd/jabstract/
Abruf: 01/2008

[29] *Homepage von JReFactory*.
http://jrefactory.sourceforge.net/
Abruf: 01/2008

[30] *Homepage von FUJABA*.
http://www.fujaba.de/
Abruf: 01/2008

[31] *JTB: Java Tree Builder*.
http://compilers.cs.ucla.edu/jtb/
Abruf: 01/2008

[32] *Homepage von GCJ: GNU Compiler for Java*.
http://gcc.gnu.org/java/
Abruf: 01/2008

[33] *Homepage von Java Espresso*.
http://www.church-project.org/Espresso/JavaEspresso.html
Abruf: 01/2008

[34] *Homepage von Eclipse*.
http://www.eclipse.org/
Abruf: 01/2008

[35] *JDT: Java Developer Tools for Eclipse*.
http://www.eclipse.org/jdt/core/index.php
Abruf: 01/2008